VINCENT LE GOFF

APPRENEZ À PROGRAMMER EN PYTHON

DÉVELOPPER EN PYTHON N'A JAMAIS ÉTÉ AUSSI FACILE !





www.siteduzero.com

VINCENT LE GOFF

APPRENEZ À PROGRAMMER EN PYTHON

DÉVELOPPER EN PYTHON N'A JAMAIS ÉTÉ AUSSI FACILE !



www.siteduzero.com

DANS LA MÊME COLLECTION





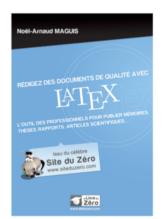
APPRENEZ À PROGRAMMER EN C

MATHIEU NEBRA
ISBN : 978-2-9535278-0-3



CONCEVEZ VOTRE SITE WEB AVEC PHP ET MYSQL

MATHIEU NEBRA
ISBN : 978-2-9535278-1-0



RÉDIGEZ DES DOCUMENTS DE QUALITÉ AVEC LATEX

NOËL-ARNAUD MAGUIS
ISBN : 978-2-9535278-4-1



REPRENEZ LE CONTRÔLE À L´AIDE DE LINUX

MATHIEU NEBRA
ISBN : 978-2-9535278-2-7



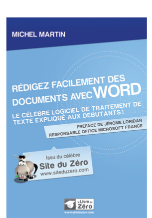
APPRENEZ À PROGRAMMER EN JAVA

CYRILLE HERBY
ISBN : 978-2-9535278-3-4



PROGRAMMEZ AVEC LE LANGAGE C++

M.NEBRA ET M.SCHALLER
ISBN : 978-2-9535278-5-8



RÉDIGEZ FACILEMENT DES DOCUMENTS AVEC WORD

MICHEL MARTIN
ISBN : 978-2-9535278-7-2

VINCENT LE GOFF

APPRENEZ À PROGRAMMER EN PYTHON

DÉVELOPPER EN PYTHON N'A JAMAIS ÉTÉ AUSSI FACILE !



www.siteduzero.com



Simple IT 2011 - ISBN : 979-10-90085-03-9

Avant-propos

J'ai commencé à m'intéresser à l'informatique, et plus particulièrement au monde de la programmation, au début du lycée, il y a maintenant plus de six ans. J'ai abordé ce terrain inconnu avec une grande curiosité… qui n'a pas encore faibli puisque je suis aujourd'hui étudiant à IN'TECH INFO, une école supérieure d'ingénierie informatique ! Au premier abord, la programmation me semblait un monde aride et froid, rempli d'équations compliquées et de notions abstraites.

Heureusement, le premier langage à avoir attiré mon attention s'est trouvé être le Python : un langage à la fois simple et extrêmement puissant, que je considère aujourd'hui comme le meilleur choix quand on souhaite apprendre à programmer. Le Python est d'ailleurs resté le langage que j'utilise le plus dans les projets libres auxquels je contribue.

Mais Python n'est pas qu'un langage simple : c'est un langage puissant. Il existe une différence entre connaître un langage et coder efficacement dans ce langage. Plusieurs années de pratiques m'ont été nécessaires pour comprendre pleinement cette différence.

Les cours sur le langage Python s'adressant aux débutants ne sont pas rares sur le Web et beaucoup sont de grande qualité. Toutefois, il en existe trop peu, à mon sens, qui permettent de comprendre à la fois la syntaxe et la philosophie du langage.

Mon objectif ici est qu'après avoir lu ce livre, vous sachiez programmer en Python. Et par « programmer », je n'entends pas seulement maîtriser la syntaxe du langage, mais aussi comprendre sa philosophie.



Étant non-voyant, je me suis efforcé de rendre ce cours aussi accessible que possible à tous. Ainsi, ne soyez pas surpris si vous y trouvez moins de schémas et d'illustrations que dans d'autres cours. J'ai fait en sorte que leur présence ne soit pas indispensable à la compréhension du lecteur.

Pour ceux qui se demandent comment je travaille, j'ai un ordinateur absolument semblable au vôtre. Pour pouvoir l'utiliser, j'installe sur mon système un logiciel qu'on appelle *lecteur d'écran*. Ce lecteur me dicte une bonne partie des informations affichées dans la fenêtre du logiciel que j'utilise, comme le navigateur Internet. Le lecteur, comme son nom l'indique, va lire grâce à une voix synthétique les informations qu'il détecte sur la fenêtre et peut également les transmettre à une *plage tactile*. C'est un

périphérique qui se charge d'afficher automatiquement en braille les informations que lui transmet le lecteur d'écran. Avec ces outils, je peux donc me servir d'un ordinateur, aller sur Internet et même programmer !

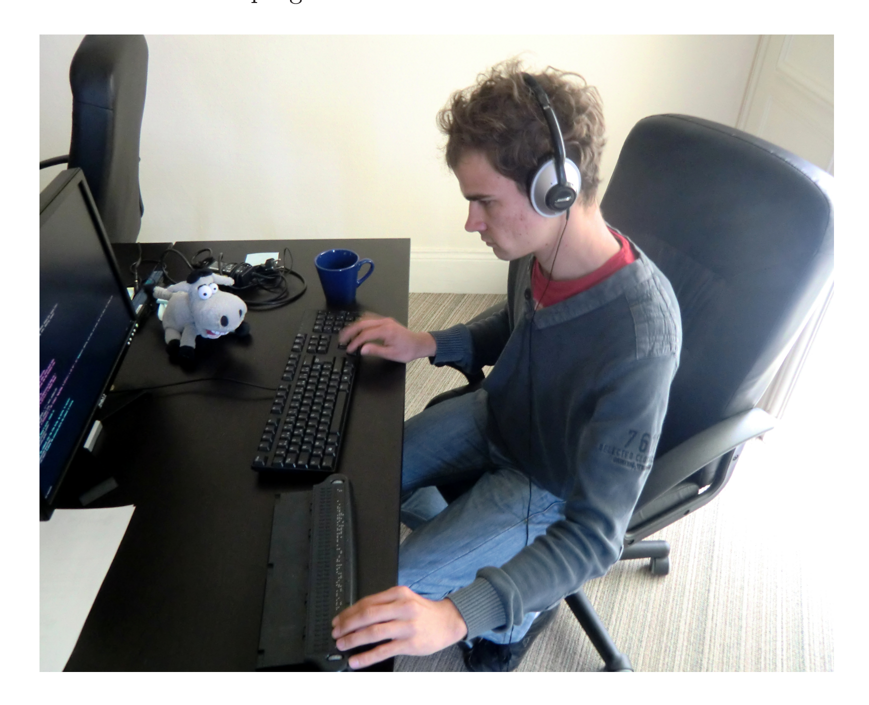


FIGURE 1 – La plage tactile et le casque transmettent les informations affichées à l'écran

Qu'allez-vous apprendre en lisant ce livre ?

Ce livre s'adresse au plus grand nombre :

– si le mot programmation ne vous évoque rien de précis, ce livre vous guidera pas à pas dans la découverte du monde du **programmeur** ;
– si vous connaissez déjà un langage de programmation autre que Python, ce livre présente de façon aussi claire que possible la syntaxe de Python et des exemples d'utilisation concrète de ce langage ;
– si vous connaissez déjà Python, ce cours peut vous servir de support comparatif avec d'autres livres et cours existants ;
– si vous enseignez le langage Python, j'ai espoir que ce livre pourra être un support utile, autant pour vous que pour vos étudiants.

Ce livre est divisé en cinq parties. Les trois premières parties sont à lire dans l'ordre, sauf si vous avez déjà de solides bases en Python :

1. **Introduction à Python** : c'est une introduction au langage de programation Python. Vous y apprendrez d'abord, si vous l'ignorez, ce que signifie **program-**

mer, ce qu'est Python et la syntaxe de base du langage.

2. **La Programmation Orientée Objet côté utilisateur** : après avoir vu les bases de Python, nous allons étudier la **façade objet** de ce langage. Dans cette partie, vous apprendrez à utiliser les **classes** que définit Python. Ne vous inquiétez pas, les concepts d'objet et de classe seront largement détaillés ici. Donc, si ces mots ne vous disent rien au premier abord, pas d'inquiétude !
3. **La Programmation Orientée Objet côté développeur** : cette partie poursuit l'approche de la façade objet débutée dans la partie précédente. Cette fois, cependant, au lieu d'être utilisateur des classes déjà définies par Python, vous allez apprendre à en créer. Là encore, ne vous inquiétez pas : nous verrons tous ces concepts pas à pas.
4. **Les merveilles de la bibliothèque standard** : cette partie étudie plus en détail certains modules déjà définis par Python. Vous y apprendrez notament à manipuler les dates et heures, faire des interfaces graphiques, construire une architecture réseau... et bien plus !
5. **Annexes** : enfin, cette partie regroupe les annexes et résumés du cours. Il s'agit de notions qui ne sont pas absolument nécessaires pour développer en Python mais que je vous encourage tout de même à lire attentivement.

Comment lire ce livre ?

Suivez l'ordre des chapitres

Lisez ce livre comme on lit un roman. Il a été conçu pour cela.

Contrairement à beaucoup de livres techniques où il est courant de lire en diagonale et de sauter certains chapitres, il est ici très fortement recommandé de suivre l'ordre du cours, à moins que vous ne soyez déjà un peu expérimentés.

Pratiquez en même temps

Pratiquez régulièrement. N'attendez pas d'avoir fini de lire ce livre pour allumer votre ordinateur et faire vos propres essais.

Utilisez les codes web !

Afin de tirer parti du Site du Zéro dont ce livre est issu, celui-ci vous propose ce qu'on appelle des « codes web ». Ce sont des codes à six chiffres à saisir sur une page du Site du Zéro pour être automatiquement redirigé vers un site web sans avoir à en recopier l'adresse.

Pour utiliser les codes web, rendez-vous sur la page suivante[1] :

1. Vous pouvez aussi utiliser le formulaire de recherche du Site du Zéro, section « Code web ».

`http://www.siteduzero.com/codeweb.html`

Un formulaire vous invite à rentrer votre code web. Faites un premier essai avec le code ci-dessous :

▷ Tester le code web
Code web : 123456

Ces codes web ont deux intérêts :

- ils vous redirigent vers les sites web présentés tout au long du cours, vous permettant ainsi d'obtenir les logiciels dans leur toute dernière version ;
- ils vous permettent de télécharger les codes sources inclus dans ce livre, ce qui vous évitera d'avoir à recopier certains programmes un peu longs.

Ce système de redirection nous permet de tenir à jour le livre que vous avez entre les mains sans que vous ayez besoin d'acheter systématiquement chaque nouvelle édition. Si un site web change d'adresse, nous modifierons la redirection mais le code web à utiliser restera le même. Si un site web disparaît, nous vous redirigerons vers une page du Site du Zéro expliquant ce qui s'est passé et vous proposant une alternative.

En clair, c'est un moyen de nous assurer de la pérennité de cet ouvrage sans que vous ayez à faire quoi que ce soit !

Remerciements

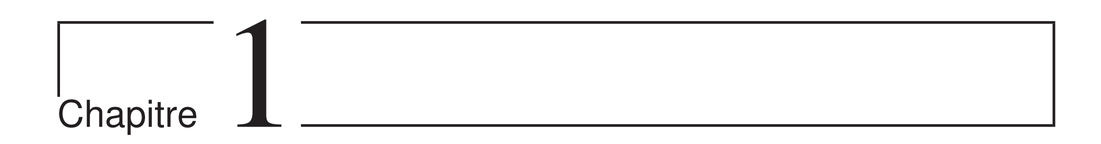
De nombreuses personnes ont, plus ou moins directement, participé à ce livre. Mes remerciements leurs sont adressés :

- à ma famille avant tout, qui a su m'encourager, dans ce projet comme dans tout autre, du début jusqu'à la fin ;
- aux personnes, trop nombreuses pour que j'en dresse ici la liste, qui ont contribué, par leurs encouragements, leurs remarques et parfois leurs critiques, à faire de ce livre ce qu'il est ;
- à l'équipe de SIMPLE IT qui a rendu ce projet envisageable et a travaillé d'arrache-pied pour qu'il se concrétise ;
- aux membres du Site du Zéro qui ont contribué à sa correction ou son enrichissement, et plus particulièrement 6pri1, Nathan21, Sergeswi et Etienne-02.

Sommaire

Première partie

Introduction à Python



Chapitre 1

Qu'est-ce que Python ?

Difficulté : 

Vous avez décidé d'apprendre le Python et je ne peux que vous en féliciter. J'essayerai d'anticiper vos questions et de ne laisser personne en arrière.

Dans ce chapitre, je vais d'abord vous expliquer ce qu'est un langage de programmation. Nous verrons ensuite brièvement l'histoire de Python, afin que vous sachiez au moins d'où vient ce langage ! Ce chapitre est théorique mais je vous conseille vivement de le lire quand même.

La dernière section portera sur l'installation de Python, une étape essentielle pour continuer ce tutoriel. Que vous travailliez avec Windows, Linux ou Mac OS X, vous y trouverez des explications précises sur l'installation.

Allez, on attaque !



Un langage de programmation ? Qu'est-ce que c'est ?

La communication humaine

Non, ceci n'est pas une explication biologique ou philosophique, ne partez pas ! Très simplement, si vous arrivez à comprendre ces suites de symboles étranges et déconcertants que sont les lettres de l'alphabet, c'est parce que nous respectons certaines conventions, dans le langage et dans l'écriture. En français, il y a des règles de grammaire et d'orthographe, je ne vous apprends rien. Vous communiquez en connaissant plus ou moins consciemment ces règles et en les appliquant plus ou moins bien, selon les cas. Cependant, ces règles peuvent être aisément contournées : personne ne peut prétendre connaître l'ensemble des règles de la grammaire et de l'orthographe françaises, et peu de gens s'en soucient. Après tout, même si vous faites des fautes, les personnes avec qui vous communiquez pourront facilement vous comprendre. Quand on communique avec un ordinateur, cependant, c'est très différent.

Mon ordinateur communique aussi !

Eh oui, votre ordinateur communique sans cesse avec vous et vous communiquez sans cesse avec lui. D'accord, il vous dit très rarement qu'il a faim, que l'été s'annonce caniculaire et que le dernier disque de ce groupe très connu était à pleurer. Il n'y a rien de magique si, quand vous cliquez sur la petite croix en haut à droite de l'application en cours, celle-ci comprend qu'elle doit se fermer.

Le langage machine

En fait, votre ordinateur se fonde aussi sur un langage pour communiquer avec vous ou avec lui-même. Les opérations qu'un ordinateur peut effectuer à la base sont des plus classiques et constituées de l'addition de deux nombres, leur soustraction, leur multiplication, leur division, entière ou non. Et pourtant, ces cinq opérations suffisent amplement à faire fonctionner les logiciels de simulation les plus complexes ou les jeux super-réalistes. Tous ces logiciels fonctionnent en gros de la même façon :

– une suite d'instructions écrites en langage machine compose le programme ;
– lors de l'exécution du programme, ces instructions décrivent à l'ordinateur ce qu'il faut faire (l'ordinateur ne peut pas le deviner).



Une liste d'instructions ? Qu'est-ce que c'est encore que cela ?

En schématisant volontairement, une instruction pourrait demander au programme de se fermer si vous cliquez sur la croix en haut à droite de votre écran, ou de rester en tâche de fond si tel est son bon plaisir. Toutefois, en langage machine, une telle action demande à elle seule un nombre assez important d'instructions. Mais bon, vous pouvez

vous en douter, parler avec l'ordinateur en langage machine, qui ne comprend que le binaire, ce n'est ni très enrichissant, ni très pratique, et en tous cas pas très marrant. On a donc inventé des langages de programmation pour faciliter la communication avec l'ordinateur.



Le langage binaire est uniquement constitué de 0 et de 1. « 01000010011011110110111001101010011011110111010101110010 », par exemple, signifie « Bonjour ». Bref, autant vous dire que discuter en binaire avec un ordinateur peut être long (surtout pour vous).

Les langages de programmation

Les langages de programmation sont des langages bien plus faciles à comprendre pour nous, pauvres êtres humains que nous sommes. Le mécanisme reste le même, mais le langage est bien plus compréhensible. Au lieu d'écrire les instructions dans une suite assez peu intelligible de 0 et de 1, les ordres donnés à l'ordinateur sont écrits dans un « langage », souvent en anglais, avec une syntaxe particulière qu'il est nécessaire de respecter. Mais avant que l'ordinateur puisse comprendre ce langage, celui-ci doit être traduit en langage machine (figure 1.1).

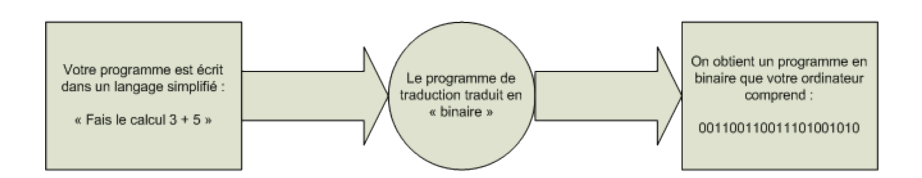


FIGURE 1.1 – Traduction d'un programme en langage binaire

En gros, le programmeur « n'a qu'à » écrire des **lignes de code** dans le langage qu'il a choisi, les étapes suivantes sont automatisées pour permettre à l'ordinateur de les décoder.

Il existe un grand nombre de langages de programmation et Python en fait partie. Il n'est pas nécessaire pour le moment de donner plus d'explications sur ces mécanismes très schématisés. Si vous n'avez pas réussi à comprendre les mots de vocabulaire et l'ensemble de ces explications, cela ne vous pénalisera pas pour la suite. Mais je trouvais intéressant de donner ces précisions quant aux façons de communiquer avec son ordinateur.

Pour la petite histoire

Python est un langage de programmation, dont la première version est sortie en 1991. Créé par **Guido van Rossum**, il a voyagé du Macintosh de son créateur, qui travaillait

à cette époque au *Centrum voor Wiskunde en Informatica* aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la **Python Software Foundation**, créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les « Monty Python ».

À quoi peut servir Python ?

Python est un langage puissant, à la fois facile à apprendre et riche en possibilités. Dès l'instant où vous l'installez sur votre ordinateur, vous disposez de nombreuses fonctionnalités intégrées au langage que nous allons découvrir tout au long de ce livre.

Il est, en outre, très facile d'étendre les fonctionnalités existantes, comme nous allons le voir. Ainsi, il existe ce qu'on appelle des **bibliothèques** qui aident le développeur à travailler sur des projets particuliers. Plusieurs bibliothèques peuvent ainsi être installées pour, par exemple, développer des interfaces graphiques en Python.

Concrètement, voilà ce qu'on peut faire avec Python :

- de petits programmes très simples, appelés **scripts**, chargés d'une mission très précise sur votre ordinateur ;
- des programmes complets, comme des jeux, des suites bureautiques, des logiciels multimédias, des clients de messagerie...
- des projets très complexes, comme des progiciels (ensemble de plusieurs logiciels pouvant fonctionner ensemble, principalement utilisés dans le monde professionnel).

Voici quelques-unes des fonctionnalités offertes par Python et ses bibliothèques :

- créer des interfaces graphiques ;
- faire circuler des informations au travers d'un réseau ;
- dialoguer d'une façon avancée avec votre système d'exploitation ;
- ... et j'en passe...

Bien entendu, vous n'allez pas apprendre à faire tout cela en quelques minutes. Mais ce cours vous donnera des bases suffisamment larges pour développer des projets qui pourront devenir, par la suite, assez importants.

Un langage de programmation interprété

Eh oui, vous allez devoir patienter encore un peu car il me reste deux ou trois choses à vous expliquer, et je suis persuadé qu'il est important de connaître un minimum ces détails qui peuvent sembler peu pratiques de prime abord. Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Les avantages d'un langage interprété sont la simplicité (on ne passe pas par une étape

de compilation avant d'exécuter son programme) et la portabilité (un langage tel que Python est censé fonctionner aussi bien sous Windows que sous Linux ou Mac OS, et on ne devrait avoir à effectuer aucun changement dans le code pour le passer d'un système à l'autre). Cela ne veut pas dire que les langages compilés ne sont pas portables, loin de là ! Mais on doit utiliser des compilateurs différents et, d'un système à l'autre, certaines instructions ne sont pas compatibles, voire se comportent différemment.

En contrepartie, un langage compilé se révélera bien plus rapide qu'un langage interprété (la traduction à la volée de votre programme ralentit l'exécution), bien que cette différence tende à se faire de moins en moins sentir au fil des améliorations. De plus, il faudra installer Python sur le système d'exploitation que vous utilisez pour que l'ordinateur puisse comprendre votre code.

Différentes versions de Python

Lors de la création de la Python Software Foundation, en 2001, et durant les années qui ont suivi, le langage Python est passé par une suite de versions que l'on a englobées dans l'appellation Python 2.x (2.3, 2.5, 2.6...). Depuis le 13 février 2009, la version 3.0.1 est disponible. Cette version casse la **compatibilité ascendante** qui prévalait lors des dernières versions.



Compatibilité quoi ?

Quand un langage de programmation est mis à jour, les développeurs se gardent bien de supprimer ou de trop modifier d'anciennes fonctionnalités. L'intérêt est qu'un programme qui fonctionne sous une certaine version marchera toujours avec la nouvelle version en date. Cependant, la Python Software Foundation, observant un bon nombre de fonctionnalités obsolètes, mises en œuvre plusieurs fois... a décidé de nettoyer tout le projet. Un programme qui tourne à la perfection sous Python 2.x devra donc être mis à jour un minimum pour fonctionner de nouveau sous Python 3. C'est pourquoi je vais vous conseiller ultérieurement de télécharger et d'installer la dernière version en date de Python. Je m'attarderai en effet sur les fonctionnalités de Python 3 et certaines d'entre elles ne seront pas accessibles (ou pas sous le même nom) dans les anciennes versions.

Ceci étant posé, tous à l'installation !

Installer Python

L'installation de Python est un jeu d'enfant, aussi bien sous Windows que sous les systèmes Unix. Quel que soit votre système d'exploitation, vous devez vous rendre sur le site officiel de Python. Pour cela, utilisez le code web suivant :

▷ Site officiel de Python
Code web : 199501

Sous Windows

1. Cliquez sur le lien `Download` dans le menu principal de la page.
2. Sélectionnez la version de Python que vous souhaitez utiliser (je vous conseille la dernière en date).
3. On vous propose un (ou plusieurs) lien(s) vers une version Windows : sélectionnez celle qui conviendra à votre processeur. Si vous avez un doute, téléchargez une version « x86 ». Si votre ordinateur vous signale qu'il ne peut exécuter le programme, essayez une autre version de Python.
4. Enregistrez puis exécutez le fichier d'installation et suivez les étapes. Ce n'est ni très long ni très difficile.
5. Une fois l'installation terminée, vous pouvez vous rendre dans le menu `Démarrer` > `Tous les programmes`. Python devrait apparaître dans cette liste (figure 1.2). Nous verrons bientôt comment le lancer, pas d'impatience...

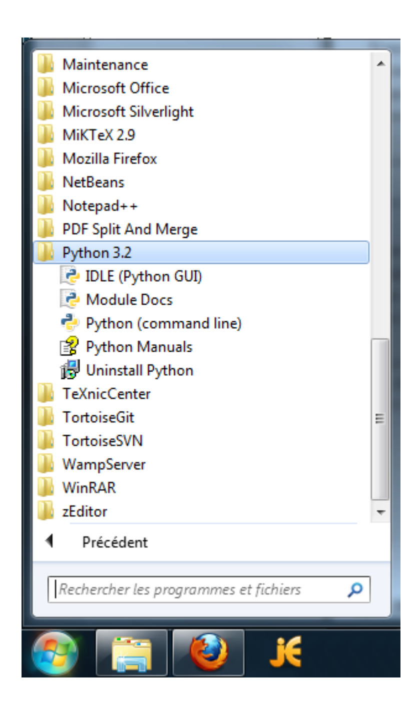


FIGURE 1.2 – Python est installé sous Windows

Sous Linux

Python est pré-installé sur la plupart des distributions Linux. Cependant, il est possible que vous n'ayez pas la dernière version en date. Pour le vérifier, tapez dans un terminal la commande `python -V`. Cette commande vous renvoie la version de Python actuellement installée sur votre système. Il est très probable que ce soit une version **2.x**, comme 2.6 ou 2.7, pour des raisons de compatibilité. Dans tous les cas, je vous conseille d'installer Python 3.x, la syntaxe est très proche de Python 2.x mais diffère quand même...

Cliquez sur `download` et téléchargez la dernière version de Python (actuellement « Python 3.2 compressed source tarball (for Linux, Unix or OS X) »). Ouvrez un terminal, puis rendez-vous dans le dossier où se trouve l'archive :

1. Décompressez l'archive en tapant : `tar -jxvf Python-3.2.tar.bz2` (cette commande est bien entendu à adapter suivant la version et le type de compression).
2. Attendez quelques instants que la décompression se termine, puis rendez-vous dans le dossier qui vient d'être créé dans le répertoire courant (`Python-3.2` dans mon cas).
3. Exécutez le script `configure` en tapant `./configure` dans la console.
4. Une fois que la configuration s'est déroulée, il n'y a plus qu'à compiler en tapant `make` puis `make altinstall`. Ces commandes ont pour but de compiler Python. La commande `make altinstall`, en particulier, crée automatiquement les liens vers la version installée. Grâce à `altinstall`, vous pouvez être sûrs que la version que vous installez n'entrera pas en conflit avec une autre déjà présente sur votre système.

Sous Mac OS X

Téléchargez la dernière version de Python. Ouvrez le fichier `.dmg` et faites un double-clic sur le paquet d'installation `Python.mpkg`

Un assistant d'installation s'ouvre, laissez-vous guider : Python est maintenant installé !

Lancer Python

Ouf ! Voilà qui est fait !

Bon, en théorie, on commence à utiliser Python dès le prochain chapitre mais, pour que vous soyez un peu récompensés de votre installation exemplaire, voici les différents moyens d'accéder à la ligne de commande Python que nous allons tout particulièrement étudier dans les prochains chapitres.

Sous Windows

Vous avez plusieurs façons d'accéder à la ligne de commande Python, la plus évidente consistant à passer par les menus `Démarrer` > `Tous les programmes` > `Python 3.2`

> `Python (Command Line)`. Si tout se passe bien, vous devriez obtenir une magnifique console (figure 1.3). Il se peut que les informations affichées dans la vôtre ne soient pas les mêmes, mais ne vous en inquiétez pas.

```
C:\Python32\python.exe
Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

FIGURE 1.3 – La console Python sous Windows



Qu'est-ce que c'est que cela ?

On verra plus tard. L'important, c'est que vous ayez réussi à ouvrir la console d'interprétation de Python, le reste attendra le prochain chapitre.

Vous pouvez également passer par la ligne de commande Windows ; à cause des raccourcis, je privilégie en général cette méthode, mais c'est une question de goût. Allez dans le menu `Démarrer`, puis cliquez sur `Exécuter`. Dans la fenêtre qui s'affiche, tapez simplement « python » et la ligne de commande Python devrait s'afficher à nouveau. Sachez que vous pouvez directement vous rendre dans `Exécuter` en tapant le raccourci `Windows` + `R`.

Pour fermer l'interpréteur de commandes Python, vous pouvez tapez « exit() » puis appuyer sur la touche `Entrée`.

Sous Linux

Lorsque vous l'avez installé sur votre système, Python a créé un lien vers l'interpréteur sous la forme **python3.X** (le X étant le numéro de la version installée).

Si, par exemple, vous avez installé Python 3.2, vous pouvez y accéder grâce à la commande :

```
$ python3.2
Python 3.2 (r32:88445, Mar  4 2011, 22:07:21)
```

```
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

Pour fermer la ligne de commande Python, n'utilisez pas CTRL + C mais CTRL + D (nous verrons plus tard pourquoi).

Sous Mac OS X

Cherchez un dossier `Python` dans le dossier `Applications`. Pour lancer Python, ouvrez l'application `IDLE` de ce dossier. Vous êtes prêts à passer au concret !

En résumé

- Python est un langage de programmation interprété, à ne pas confondre avec un langage compilé.
- Il permet de créer toutes sortes de programmes, comme des jeux, des logiciels, des progiciels, etc.
- Il est possible d'associer des **bibliothèques** à Python afin d'étendre ses possibilités.
- Il est portable, c'est à dire qu'il peut fonctionner sous différents systèmes d'exploitation (Windows, Linux, Mac OS X,...).

Chapitre 2

Premiers pas avec l'interpréteur de commandes Python

Difficulté : 

Après les premières notions théoriques et l'installation de Python, il est temps de découvrir un peu l'interpréteur de commandes de ce langage. Même si ces petits tests vous semblent anodins, vous découvrirez dans ce chapitre les premiers rudiments de la syntaxe du langage et je vous conseille fortement de me suivre pas à pas, surtout si vous êtes face à votre premier langage de programmation.

Comme tout langage de programmation, Python a une syntaxe claire : on ne peut pas lui envoyer n'importe quelle information dans n'importe quel ordre. Nous allons voir ici ce que Python mange. . . et ce qu'il ne mange pas.



Où est-ce qu'on est, là ?

Pour commencer, je vais vous demander de retourner dans l'interpréteur de commandes Python (je vous ai montré, à la fin du chapitre précédent, comment y accéder en fonction de votre système d'exploitation).

Je vous rappelle les informations qui figurent dans cette fenêtre, même si elles peuvent être différentes chez vous en fonction de votre version et de votre système d'exploitation.

```
Python 3.2.1 (default, Jul 10 2011, 21:51:15) [MSC v.1500 32
    bit (Intel)] on win 32
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

À sa façon, Python vous souhaite la bienvenue dans son interpréteur de commandes.



Attends, attends. C'est quoi cet interpréteur ?

Souvenez-vous, au chapitre précédent, je vous ai donné une brève explication sur la différence entre langages compilés et langages interprétés. Eh bien, cet interpréteur de commandes va nous permettre de tester directement du code. Je saisis une ligne d'instructions, j'appuie sur la touche Entrée de mon clavier, je regarde ce que me répond Python (s'il me dit quelque chose), puis j'en saisis une deuxième, une troisième... Cet interpréteur est particulièrement utile pour comprendre les bases de Python et réaliser nos premiers petits programmes. Le principal inconvénient, c'est que le code que vous saisissez n'est pas sauvegardé (sauf si vous l'enregistrez manuellement, mais chaque chose en son temps).

Dans la fenêtre que vous avez sous les yeux, l'information qui ne change pas d'un système d'exploitation à l'autre est la série de trois chevrons qui se trouve en bas à gauche des informations : >>>. Ces trois signes signifient : « je suis prêt à recevoir tes instructions ».

Comme je l'ai dit, les langages de programmation respectent une syntaxe claire. Vous ne pouvez pas espérer que l'ordinateur comprenne si, dans cette fenêtre, vous commencez par lui demander : « j'aimerais que tu me codes un jeu vidéo génial ». Et autant que vous le sachiez tout de suite (bien qu'à mon avis, vous vous en doutiez), on est très loin d'obtenir des résultats aussi spectaculaires à notre niveau.

Tout cela pour dire que, si vous saisissez n'importe quoi dans cette fenêtre, la probabilité est grande que Python vous indique, clairement et fermement, qu'il n'a rien compris.

Si, par exemple, vous saisissez « premier test avec Python », vous obtenez le résultat suivant :

```
>>> premier test avec Python
File "<stdin>", line 1
```

```
premier test avec Python
^
SyntaxError: invalid syntax
>>>
```

Eh oui, l'interpréteur parle en anglais et les instructions que vous saisirez, comme pour l'écrasante majorité des langages de programmation, seront également en anglais. Mais pour l'instant, rien de bien compliqué : l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction. Il vous indique le numéro de la ligne (en l'occurence la première), qu'il vous répète obligeamment (ceci est très utile quand on travaille sur un programme de plusieurs centaines de lignes). Puis il vous dit ce qui l'arrête, ici : `SyntaxError: invalid syntax`. Limpide n'est-ce pas ? Ce que vous avez saisi est incompréhensible pour Python. Enfin, la preuve qu'il n'est pas rancunier, c'est qu'il vous affiche à nouveau une série de trois chevrons, montrant bien qu'il est prêt à retenter l'aventure.

Bon, c'est bien joli de recevoir un message d'erreur au premier test mais je me doute que vous aimeriez bien voir des trucs qui fonctionnent, maintenant. C'est parti donc.

Vos premières instructions : un peu de calcul mental pour l'ordinateur

C'est assez trivial, quand on y pense, mais je trouve qu'il s'agit d'une excellente manière d'aborder pas à pas la syntaxe de Python. Nous allons donc essayer d'obtenir les résultats de calculs plus ou moins compliqués. Je vous rappelle encore une fois qu'exécuter les tests en même temps que moi sur votre machine est une très bonne façon de vous rendre compte de la syntaxe et surtout, de la retenir.

Saisir un nombre

Vous avez pu voir sur notre premier (et à ce jour notre dernier) test que Python n'aimait pas particulièrement les suites de lettres qu'il ne comprend pas. Par contre, l'interpréteur adore les nombres. D'ailleurs, il les accepte sans sourciller, sans une seule erreur :

```
>>> 7
7
>>>
```

D'accord, ce n'est pas extraordinaire. On saisit un nombre et l'interpréteur le renvoie. Mais dans bien des cas, ce simple retour indique que l'interpréteur a bien compris et que votre saisie est en accord avec sa syntaxe. De même, vous pouvez saisir des nombres à virgule.

```
>>> 9.5
```

```
9.5
>>>
```



Attention : on utilise ici la notation anglo-saxonne, c'est-à-dire que le point remplace la virgule. La virgule a un tout autre sens pour Python, prenez donc cette habitude dès maintenant.

Il va de soi que l'on peut tout aussi bien saisir des nombres négatifs (vous pouvez d'ailleurs faire l'essai).

Opérations courantes

Bon, il est temps d'apprendre à utiliser les principaux opérateurs de Python, qui vont vous servir pour la grande majorité de vos programmes.

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
7
>>> -2 + 93
91
>>> 9.5 + 2
11.5
>>> 3.11 + 2.08
5.1899999999999995
>>>
```



Pourquoi ce dernier résultat approximatif ?

Python n'y est pas pour grand chose. En fait, le problème vient en grande partie de la façon dont les nombres à virgule sont écrits dans la mémoire de votre ordinateur. C'est pourquoi, en programmation, on préfère travailler autant que possible avec des nombres entiers. Cependant, vous remarquerez que l'erreur est infime et qu'elle n'aura pas de réel impact sur les calculs. Les applications qui ont besoin d'une précision mathématique à toute épreuve essayent de pallier ces défauts par d'autres moyens mais ici, ce ne sera pas nécessaire.

Faites également des tests pour la soustraction, la multiplication et la division : il n'y a rien de difficile.

Division entière et modulo

Si vous avez pris le temps de tester la division, vous vous êtes rendu compte que le résultat est donné avec une virgule flottante.

```
>>> 10 / 5
2.0
>>> 10 / 3
3.3333333333333335
>>>
```

Il existe deux autres opérateurs qui permettent de connaître le résultat d'une division entière et le reste de cette division.

Le premier opérateur utilise le symbole « // ». Il permet d'obtenir la partie entière d'une division.

```
>>> 10 // 3
3
>>>
```

L'opérateur « % », que l'on appelle le « modulo », permet de connaître le reste de la division.

```
>>> 10%3
1
>>>
```

Ces notions de *partie entière* et de *reste de division* ne sont pas bien difficiles à comprendre et vous serviront très probablement par la suite.

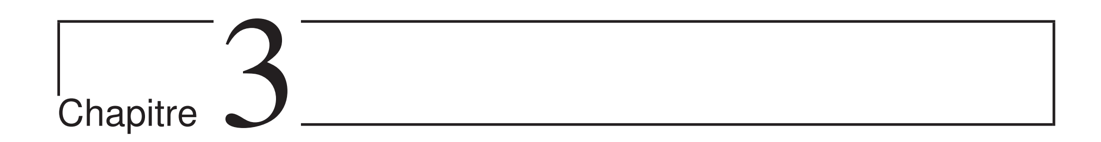
Si vous avez du mal à en saisir le sens, sachez donc que :

- La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurence, 3).
- Pour obtenir le modulo d'une division, on « récupère » son reste. Dans notre exemple, $10/3 = 3$ et il reste 1. Une fois que l'on a compris cela, ce n'est pas bien compliqué.

Souvenez-vous bien de ces deux opérateurs, et surtout du modulo « % », dont vous aurez besoin dans vos programmes futurs.

En résumé

- L'interpréteur de commandes Python permet de tester du code au fur et à mesure qu'on l'écrit.
- L'interpréteur Python accepte des nombres et est capable d'effectuer des calculs.
- Un nombre décimal s'écrit avec un point et non une virgule.
- Les calculs impliquant des nombres décimaux donnent parfois des résultats approximatifs, c'est pourquoi on préfèrera, dans la mesure du possible, travailler avec des nombres entiers.


Chapitre 3

Le monde merveilleux des variables

Difficulté : 

Au chapitre précédent, vous avez saisi vos premières instructions en langage Python, bien que vous ne vous en soyez peut-être pas rendu compte. Il est également vrai que les instructions saisies auraient fonctionné dans la plupart des langages. Ici, cependant, nous allons commencer à approfondir un petit peu la syntaxe du langage, tout en découvrant un concept important de la programmation : les variables.

Ce concept est essentiel et vous ne pouvez absolument pas faire l'impasse dessus. Mais je vous rassure, il n'y a rien de compliqué, que de l'utile et de l'agréable.



C'est quoi, une variable ? Et à quoi cela sert-il ?

Les variables sont l'un des concepts qui se retrouvent dans la majorité (et même, en l'occurrence, la totalité) des langages de programmation. Autant dire que sans variable, on ne peut pas programmer, et ce n'est pas une exagération.

C'est quoi, une variable ?

Une variable est une donnée de votre programme, stockée dans votre ordinateur. C'est un code alpha-numérique que vous allez lier à une donnée de votre programme, afin de pouvoir l'utiliser à plusieurs reprises et faire des calculs un peu plus intéressants avec. C'est bien joli de savoir faire des opérations mais, si on ne peut pas stocker le résultat quelque part, cela devient très vite ennuyeux.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

Comment cela fonctionne-t-il ?

Le plus simplement du monde. Vous allez dire à Python : « je veux que, dans une variable que je nomme `age`, tu stockes mon âge, pour que je puisse le retenir (si j'ai la mémoire très courte), l'augmenter (à mon anniversaire) et l'afficher si besoin est ».

Comme je vous l'ai dit, on ne peut pas passer à côté des variables. Vous ne voyez peut-être pas encore tout l'intérêt de stocker des informations de votre programme et pourtant, si vous ne stockez rien, vous ne pouvez pratiquement rien faire.

En Python, pour donner une valeur à une variable, il suffit d'écrire `nom_de_la_variable = valeur`.

Une variable doit respecter quelques règles de syntaxe incontournables :

1. Le nom de la variable ne peut être composé que de lettres, majuscules ou minuscules, de chiffres et du symbole souligné « _ » (appelé *underscore* en anglais).
2. Le nom de la variable ne peut pas commencer par un chiffre.
3. Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable `AGE` est différente de `aGe`, elle-même différente de `age`).

Au-delà de ces règles de syntaxe incontournables, il existe des conventions définies par les programmeurs eux-mêmes. L'une d'elle, que j'ai tendance à utiliser assez souvent, consiste à écrire la variable en minuscules et à remplacer les espaces éventuels par un espace souligné « _ ». Si je dois créer une variable contenant mon âge, elle se nommera donc `mon_age`. Une autre convention utilisée consiste à passer en majuscule le premier caractère de chaque mot, à l'exception du premier mot constituant la variable. La variable contenant mon âge se nommerait alors `monAge`.

Vous pouvez utiliser la convention qui vous plaît, ou même en créer une bien à vous, mais essayez de rester cohérent et de n'utiliser qu'une seule convention d'écriture. En effet, il est essentiel de pouvoir vous repérer dans vos variables dès que vous commencez à travailler sur des programmes volumineux.

Ainsi, si je veux associer mon âge à une variable, la syntaxe sera :

```
mon_age = 21
```

L'interpréteur vous affiche aussitôt trois chevrons sans aucun message. Cela signifie qu'il a bien compris et qu'il n'y a eu aucune erreur.

Sachez qu'on appelle cette étape *l'affectation de valeur à une variable* (parfois raccourci en « affectation de variable »). On dit en effet qu'on a affecté la valeur 21 à la variable `mon_age`.

On peut afficher la valeur de cette variable en la saisissant simplement dans l'interpréteur de commandes.

```
>>> mon_age
21
>>>
```



Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs. Je les mets pour des raisons de lisibilité.



Bon, c'est bien joli tout cela, mais qu'est-ce qu'on fait avec cette variable ?

Eh bien, tout ce que vous avez déjà fait au chapitre précédent, mais cette fois en utilisant la variable comme un nombre à part entière. Vous pouvez même affecter à d'autres variables des valeurs obtenues en effectuant des calculs sur la première et c'est là toute la puissance de ce mécanisme.

Essayons par exemple d'augmenter de 2 la variable `mon_age`.

```
>>> mon_age = mon_age + 2
>>> mon_age
23
>>>
```

Encore une fois, lors de l'affectation de la valeur, rien ne s'affiche, ce qui est parfaitement normal.

Maintenant, essayons d'affecter une valeur à une autre variable d'après la valeur de `mon_age`.

```
>>> mon_age_x2 = mon_age * 2
```

```
>>> mon_age_x2
46
>>>
```

Encore une fois, je vous invite à tester en long, en large et en travers cette possibilité. Le concept n'est pas compliqué mais extrêmement puissant. De plus, comparé à certains langages, affecter une valeur à une variable est extrêmement simple. Si la variable n'est pas créée, Python s'en charge automatiquement. Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle. Quoi de plus simple ?



Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

En voici la liste pour Python 3 :

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Ces mots-clés sont utilisés par Python, vous ne pouvez pas construire de variables portant ces noms. Vous allez découvrir dans la suite de ce cours la majorité de ces mots-clés et comment ils s'utilisent.

Les types de données en Python

Là se trouve un concept très important, que l'on retrouve dans beaucoup de langages de programmation. Ouvrez grand vos oreilles, ou plutôt vos yeux, car vous devrez être parfaitement à l'aise avec ce concept pour continuer la lecture de ce livre. Rassurez-vous toutefois, du moment que vous êtes attentifs, il n'y a rien de compliqué à comprendre.

Qu'entend-on par « type de donnée » ?

Jusqu'ici, vous n'avez travaillé qu'avec des nombres. Et, s'il faut bien avouer qu'on ne fera que très rarement un programme sans aucun nombre, c'est loin d'être la seule donnée que l'on peut utiliser en Python. À terme, vous serez même capables de créer vos propres types de données, mais n'anticipons pas.

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec. Dans ce chapitre, vous allez apprendre à travailler avec

des chaînes de caractères, et multiplier une chaîne de caractères ne se fait pas du tout comme la multiplication d'un nombre. Pour certains types de données, la multiplication n'a d'ailleurs aucun sens. Python associe donc à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée en particulier.

Les différents types de données

Nous n'allons voir ici que les incontournables et les plus faciles à manier. Des chapitres entiers seront consacrés aux types plus complexes.

Les nombres entiers

Et oui, Python différencie les entiers des nombres à virgule flottante !



Pourquoi cela ?

Initialement, c'est surtout pour une question de place en mémoire mais, pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité de nos jours.

Le type entier se nomme `int` en Python (qui correspond à l'anglais « integer », c'est-à-dire entier). La forme d'un entier est un nombre sans virgule.

```
3
```

Nous avons vu au chapitre précédent les opérations que l'on pouvait effectuer sur ce type de données et, même si vous ne vous en souvenez pas, les deviner est assez élémentaire.

Les nombres flottants

Les flottants sont les nombres à virgule. Ils se nomment `float` en Python (ce qui signifie « flottant » en anglais). La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point). Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui ajouter une partie flottante de 0 (exemple **52.0**).

```
3.152
```

Les nombres après la virgule ne sont pas infinis, puisque rien n'est infini en informatique. Mais la précision est assez importante pour travailler sur des données très fines.

Les chaînes de caractères

Heureusement, les types de données disponibles en Python ne sont pas limités aux seuls nombres, bien loin de là. Le dernier type « simple » que nous verrons dans ce chapitre

est la chaîne de caractères. Ce type de donnée permet de stocker une série de lettres, pourquoi pas une phrase.

On peut écrire une chaîne de caractères de différentes façons :

- entre guillemets (`"ceci est une chaîne de caractères"`) ;
- entre apostrophes (`'ceci est une chaîne de caractères'`) ;
- entre triples guillemets (`"""ceci est une chaîne de caractères"""`).

On peut, à l'instar des nombres (et de tous les types de données) stocker une chaîne de caractères dans une variable (`ma_chaine = "Bonjour, la foule !"`)

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez `chaine = 'J'aime le Python!'`, vous obtenez le message suivant :

```
File "<stdin>", line 1
chaine = 'J'aime le Python!'
^
SyntaxError: invalid syntax
```

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà. Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi un caractère anti-slash « \ » avant les apostrophes contenues dans le message.

```
chaine = 'J\'aime le Python!'
```

On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs.

```
chaine2 = "\"Le seul individu formé, c'est celui qui a appris
    comment apprendre (...)\" (Karl Rogers, 1976)"
```

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne (`"essai\nsur\nplusieurs\nlignes"`). Pour écrire un véritable anti-slash dans une chaîne, il faut l'échapper lui-même (et donc écrire « \\ »).



L'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ». Nous verrons dans la partie suivante comment afficher réellement ces chaînes de caractères et pourquoi l'interpréteur ne les affiche pas comme il le devrait.

Utiliser les triples guillemets pour encadrer une chaîne de caractères dispense d'échapper les guillemets et apostrophes, et permet d'écrire plusieurs lignes sans symboliser les retours à la ligne au moyen de « \n ».

```
>>> chaine3 = """Ceci est un nouvel
... essai sur plusieurs
... lignes"""
>>>
```

Notez que les trois chevrons sont remplacés par trois points : cela signifie que l'interpréteur considère que vous n'avez pas fini d'écrire cette instruction. En effet, celle-ci ne s'achève qu'une fois la chaîne refermée avec trois nouveaux guillemets. Les sauts de lignes seront automatiquement remplacés, dans la chaîne, par des « \n ».

Vous pouvez utiliser, à la place des trois guillemets, trois apostrophes qui jouent exactement le même rôle. Je n'utilise personnellement pas ces délimiteurs, mais sachez qu'ils existent et ne soyez pas surpris si vous les voyez un jour dans un code source.

Voilà, nous avons bouclé le rapide tour d'horizon des types simples. Qualifier les chaînes de caractères de type simple n'est pas strictement vrai mais nous n'allons pas, dans ce chapitre, entrer dans le détail des opérations que l'on peut effectuer sur ces chaînes. C'est inutile pour l'instant et ce serait hors sujet. Cependant, rien ne vous empêche de tester vous mêmes quelques opérations comme l'addition et la multiplication (dans le pire des cas, Python vous dira qu'il ne peut pas faire ce que vous lui demandez et, comme nous l'avons vu, il est peu rancunier).

Un petit bonus

Au chapitre précédent, nous avons vu les opérateurs « classiques » pour manipuler des nombres mais aussi, comme on le verra plus tard, d'autres types de données. D'autres opérateurs ont été créés afin de simplifier la manipulation des variables.

Vous serez amenés par la suite, et assez régulièrement, à incrémenter des variables. L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre. Jusqu'ici, j'ai procédé comme ci-dessous pour augmenter une variable de 1 :

```
variable = variable + 1
```

Cette syntaxe est claire et intuitive mais assez longue, et les programmeurs, tout le monde le sait, sont des fainéants nés. On a donc trouvé plus court.

```
variable += 1
```

L'opérateur += revient à ajouter à la variable la valeur qui suit l'opérateur. Les opérateurs -=, *= et /= existent également, bien qu'ils soient moins utilisés.

Quelques trucs et astuces pour vous faciliter la vie

Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable qui retient l'une des deux valeurs… ici c'est bien plus simple :

```
>>> a = 5
```

```
2 >>> b = 32
3 >>> a,b = b,a # permutation
4 >>> a
5 32
6 >>> b
7 5
8 >>>
```

Comme vous le voyez, après l'exécution de la ligne 3, les variables `a` et `b` ont échangé leurs valeurs. On retrouvera cette distribution d'affectation bien plus loin.

On peut aussi affecter assez simplement une même valeur à plusieurs variables :

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

Enfin, ce n'est pas encore d'actualité pour vous mais sachez qu'on peut couper une instruction Python, pour l'écrire sur deux lignes ou plus.

```
>>> 1 + 4 - 3 * 19 + 33 - 45 * 2 + (8 - 3) \
... -6 + 23.5
-86.5
>>>
```

Comme vous le voyez, le symbole « \ » permet, avant un saut de ligne, d'indiquer à Python que « cette instruction se poursuit à la ligne suivante ». Vous pouvez ainsi morceler votre instruction sur plusieurs lignes.

Première utilisation des fonctions

Eh bien, tout cela avance gentiment. Je me permets donc d'introduire ici, dans ce chapitre sur les variables, l'utilisation des fonctions. Il s'agit finalement bien davantage d'une application concrète de ce que vous avez appris à l'instant. Un chapitre entier sera consacré aux fonctions, mais utiliser celles que je vais vous montrer n'est pas sorcier et pourra vous être utile.

Utiliser une fonction



À quoi servent les fonctions ?

Une fonction exécute un certain nombre d'instructions déjà enregistrées. En gros, c'est comme si vous enregistriez un groupe d'instructions pour faire une action précise et que vous lui donniez un nom. Vous n'avez plus ensuite qu'à appeler cette fonction par son nom autant de fois que nécessaire (cela évite bon nombre de répétitions). Mais nous verrons tout cela plus en détail par la suite.

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée ; ces paramètres sont des informations que vous passez à la fonction afin qu'elle travaille dessus. Les fonctions que je vais vous montrer ne font pas exception. Ce concept vous semble peut-être un peu difficile à saisir dans son ensemble mais rassurez-vous, les exemples devraient tout rendre limpide.

Les fonctions s'utilisent en respectant la syntaxe suivante :
`nom_de_la_fonction(parametre_1,parametre_2,...,parametre_n)`.

- Vous commencez par écrire le nom de la fonction.
- Vous placez entre parenthèses les paramètres de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction « type »

Dans la partie précédente, je vous ai présenté les types de données simples, du moins une partie d'entre eux. Une des grandes puissances de Python est qu'il comprend automatiquement de quel type est une variable et cela lors de son affectation. Mais il est pratique de pouvoir savoir de quel type est une variable.

La syntaxe de cette fonction est simple :

```
type(nom_de_la_variable)
```

La fonction renvoie le type de la variable passée en paramètre. Vu que nous sommes dans l'interpréteur de commandes, cette valeur sera affichée. Si vous saisissez dans l'interpréteur les lignes suivantes :

```
>>> a = 3
>>> type(a)
```

Vous obtenez :

```
<class 'int'>
```

Python vous indique donc que la variable `a` appartient à la classe des entiers. Cette notion de classe ne sera pas approfondie avant bien des chapitres mais sachez qu'on peut la rapprocher d'un type de donnée.

Vous pouvez faire le test sans passer par des variables :

```
>>> type(3.4)
<class 'float'>
```

```
>>> type("un essai")
<class 'str'>
>>>
```



str est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

La fonction print

La fonction `print` permet d'afficher la valeur d'une ou plusieurs variables.



Mais... on ne fait pas exactement la même chose en saisissant juste le nom de la variable dans l'interpréteur ?

Oui et non. L'interpréteur affiche bien la valeur de la variable car il affiche automatiquement tout ce qu'il peut, pour pouvoir suivre les étapes d'un programme. Cependant, quand vous ne travaillerez plus avec l'interpréteur, taper simplement le nom de la variable n'aura aucun effet. De plus, et vous l'aurez sans doute remarqué, l'interpréteur entoure les chaînes de caractères de délimiteurs et affiche les caractères d'échappement, tout ceci encore pour des raisons de clarté.

La fonction `print` est dédiée à l'affichage uniquement. Le nombre de ses paramètres est variable, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables. Considérez cet exemple :

```
>>> a = 3
>>> print(a)
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
```

Le premier *appel* à `print` se contente d'afficher la valeur de la variable `a`, c'est-à-dire « 3 ». Le second appel à `print` affiche :

```
a = 6 et b = 4
```

Ce deuxième appel à `print` est peut-être un peu plus dur à comprendre. En fait, on passe quatre paramètres à `print`, deux chaînes de caractères et les variables `a` et `b`. Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Relisez bien cet exemple, il montre tout l'intérêt des fonctions. Si vous avez du mal à le comprendre dans son ensemble, décortiquez-le en prenant indépendamment chaque paramètre.

Testez l'utilisation de `print` avec d'autres types de données et en insérant des chaînes avec des sauts de lignes et des caractères échappés, pour bien vous rendre compte de la différence.

Un petit « Hello World ! » ?

Quand on fait un cours sur un langage, quel qu'il soit, il est d'usage de présenter le programme « Hello World ! », qui illustre assez rapidement la syntaxe superficielle d'un langage.

Le but du jeu est très simple : écrire un programme qui affiche « Hello World ! » à l'écran. Dans certains langages, notamment les langages compilés, vous pourrez nécessiter jusqu'à une dizaine de lignes pour obtenir ce résultat. En Python, comme nous venons de le voir, il suffit d'une seule ligne :

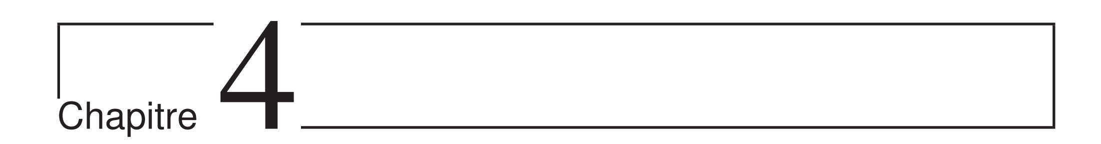
```
>>> print("Hello World !")
```

Pour plus d'informations, n'hésitez pas à consulter la page Wikipédia consacrée à « Hello World ! » ; vous avez même des codes rédigés en différents langages de programmation, cela peut être intéressant.

▷ Wikipédia - « Hello World ! »
Code web : 696192

En résumé

- Les variables permettent de conserver dans le temps des données de votre programme.
- Vous pouvez vous servir de ces variables pour différentes choses : les afficher, faire des calculs avec, etc.
- Pour affecter une valeur à une variable, on utilise la syntaxe `nom_de_variable = valeur`.
- Il existe différents types de variables, en fonction de l'information que vous désirez conserver : `int`, `float`, chaîne de caractères etc.
- Pour afficher une donnée, comme la valeur d'une variable par exemple, on utilise la fonction `print`.



Les structures conditionnelles

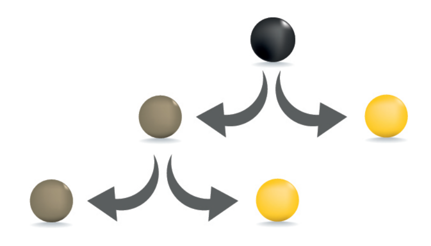
Difficulté : 

Jusqu'à présent, nous avons testé des instructions d'une façon linéaire : l'interpréteur exécutait au fur et à mesure le code que vous saisissiez dans la console. Mais nos programmes seraient bien pauvres si nous ne pouvions, de temps à autre, demander à exécuter certaines instructions dans un cas, et d'autres instructions dans un autre cas.

Dans ce chapitre, je vais vous parler des structures conditionnelles, qui vont vous permettre de faire des tests et d'aller plus loin dans la programmation.

Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

Vous finirez ce chapitre en créant votre premier « vrai » programme : même si vous ne pensez pas encore pouvoir faire quelque chose de très consistant, à la fin de ce chapitre vous aurez assez de matière pour coder un petit programme dans un but très précis.



Vos premières conditions et blocs d'instructions

Forme minimale en `if`

Les conditions sont un concept essentiel en programmation (oui oui, je me répète à force mais il faut avouer que des concepts essentiels, on n'a pas fini d'en voir). Elles vont vous permettre de faire une action précise si, par exemple, une variable est positive, une autre action si cette variable est négative, ou une troisième action si la variable est nulle. Comme un bon exemple vaut mieux que plusieurs lignes d'explications, voici un exemple clair d'une condition prise sous sa forme la plus simple.



Dès à présent dans mes exemples, j'utiliserai des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code (car, vous vous en rendrez compte, relire ses programmes après plusieurs semaines d'abandon, sans commentaire, ce peut être parfois plus qu'ardu). En Python, un commentaire débute par un dièse (« # ») et se termine par un saut de ligne. Tout ce qui est compris entre ce # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le # en début de ligne) ou une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Cela étant posé, revenons à nos conditions :

```
1 >>> # Premier exemple de condition
2 >>> a = 5
3 >>> if a > 0: # Si a est supérieur à 0
4 ...     print("a est supérieur à 0.")
5 ...
6 a est supérieur à 0.
7 >>>
```

Détaillons ce code, ligne par ligne :

1. La première ligne est un commentaire décrivant qu'il s'agit du premier test de condition. Elle est ignorée par l'interpréteur et sert juste à vous renseigner sur le code qui va suivre.
2. Cette ligne, vous devriez la comprendre sans aucune aide. On se contente d'affecter la valeur 5 à la variable `a`.
3. Ici se trouve notre test conditionnel. Il se compose, dans l'ordre :
   - du mot clé `if` qui signifie « si » en anglais ;
   - de la condition proprement dite, `a > 0`, qu'il est facile de lire (une liste des opérateurs autorisés pour la comparaison sera présentée plus bas) ;
   - du signe deux points, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'omettez.

4. Ici se trouve l'instruction à exécuter dans le cas où `a` est supérieur à `0`. Après que vous ayez appuyé sur `Entrée` à la fin de la ligne précédente, l'interpréteur vous présente la série de trois points qui signifie qu'il attend la saisie du **bloc d'instructions** concerné avant de l'interpréter. Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée, c'est-à-dire décalée vers la droite. Des explications supplémentaires seront données un peu plus bas sur les indentations.
5. L'interpréteur vous affiche à nouveau la série de trois points et vous pouvez en profiter pour saisir une nouvelle instruction dans ce bloc d'instructions. Ce n'est pas le cas pour l'instant. Vous appuyez donc sur `Entrée` sans avoir rien écrit et l'interpréteur vous affiche le message « a est supérieur à 0 », ce qui est assez logique vu que `a` est effectivement supérieur à `0`.

Il y a deux notions importantes sur lesquelles je dois à présent revenir, elles sont complémentaires ne vous en faites pas.

La première est celle de bloc d'instructions. On entend par bloc d'instructions une série d'instructions qui s'exécutent dans un cas précis (par condition, comme on vient de le voir, par répétition, comme on le verra plus tard...). Ici, notre bloc n'est constitué que d'une seule instruction (la ligne 4 qui fait appel à `print`). Mais rien ne vous empêche de mettre plusieurs instructions dans ce bloc.

```
a = 5
b = 8
if a > 0:
    # On incrémente la valeur de b
    b += 1
    # On affiche les valeurs des variables
    print("a =",a,"et b =",b)
```

La seconde notion importante est celle d'indentation. On entend par indentation un certain décalage vers la droite, obtenu par un (ou plusieurs) espaces ou tabulations.

Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java, d'un confort de lecture mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Forme complète (`if`, `elif` et `else`)

Les limites de la condition simple en `if`

La première forme de condition que l'on vient de voir est pratique mais assez incomplète.

Considérons, par exemple, une variable `a` de type entier. On souhaite faire une action si cette variable est positive et une action différente si elle est négative. Il est possible d'obtenir ce résultat avec la forme simple d'une condition :

```
>>> a = 5
>>> if a > 0: # Si a est positif
...     print("a est positif.")
... if a < 0: # a est négatif
...     print("a est négatif.")
```

Amusez-vous à changer la valeur de `a` et exécutez à chaque fois les conditions ; vous obtiendrez des messages différents, sauf si `a` est égal à 0. En effet, aucune action n'a été prévue si a vaut 0.

Cette méthode n'est pas optimale, tout d'abord parce qu'elle nous oblige à écrire deux conditions séparées pour tester une même variable. De plus, et même si c'est dur à concevoir par cet exemple, dans le cas où la variable remplirait les deux conditions (ici c'est impossible bien entendu), les deux portions de code s'exécuteraient.

La condition `if` est donc bien pratique mais insuffisante.

L'instruction `else` :

Le mot-clé `else`, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction `if`.

```
>>> age = 21
>>> if age >= 18: # Si age est supérieur ou égal à 18
...     print("Vous êtes majeur.")
... else: # Sinon (age inférieur à 18)
... print("Vous êtes mineur.")
```

Je pense que cet exemple suffit amplement à exposer l'utilisation de `else`. La seule subtilité est de bien se rendre compte que Python exécute soit l'un, soit l'autre, et jamais les deux. Notez que cette instruction `else` doit se trouver au même niveau d'indentation que l'instruction `if` qu'elle complète. De plus, elle se termine également par deux points puisqu'il s'agit d'une condition, même si elle est sous-entendue.

L'exemple de tout à l'heure pourrait donc se présenter comme suit, avec l'utilisation de `else` :

```
>>> a = 5
>>> if a > 0:
...     print("a est supérieur à 0.")
... else:
...     print("a est inférieur ou égal à 0.")
```



Mais... le résultat n'est pas tout à fait le même, si ?

Non, en effet. Vous vous rendrez compte que, cette fois, le cas où `a` vaut `0` est bien pris en compte. En effet, la condition initiale prévoit d'exécuter le premier bloc d'instructions si `a` est strictement supérieur à `0`. Sinon, on exécute le second bloc d'instructions.

Si l'on veut faire la différence entre les nombres positifs, négatifs et nuls, il va falloir utiliser une condition intermédiaire.

L'instruction `elif` :

Le mot clé `elif` est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ». Dans l'exemple que nous venons juste de voir, l'idéal serait d'écrire :

– si `a` est strictement supérieur à `0`, on dit qu'il est positif ;
– sinon si `a` est strictement inférieur à `0`, on dit qu'il est négatif ;
– sinon, (`a` ne peut qu'être égal à `0`), on dit alors que `a` est nul.

Traduit en langage Python, cela donne :

```
>>> if a > 0: # Positif
...     print("a est positif.")
... elif a < 0: # Négatif
...     print("a est négatif.")
... else: # Nul
    print("a est nul.")
```

De même que le `else`, le `elif` est sur le même niveau d'indentation que le `if` initial. Il se termine aussi par deux points. Cependant, entre le `elif` et les deux points se trouve une nouvelle condition. Linéairement, le schéma d'exécution se traduit comme suit :

1. On regarde si `a` est strictement supérieur à `0`. Si c'est le cas, on affiche « a est positif » et on s'arrête là.
2. Sinon, on regarde si `a` est strictement inférieur à `0`. Si c'est le cas, on affiche « a est négatif » et on s'arrête.
3. Sinon, on affiche « a est nul ».



Attention : quand je dis « on s'arrête », il va de soi que c'est uniquement pour cette condition. S'il y a du code après les trois blocs d'instructions, il sera exécuté dans tous les cas.

Vous pouvez mettre autant de `elif` que vous voulez après une condition en `if`. Tout comme le `else`, cette instruction est facultative et, quand bien même vous construiriez une instruction en `if`, `elif`, vous n'êtes pas du tout obligé de prévoir un `else` après. En revanche, l'instruction `else` ne peut figurer qu'une fois, clôturant le bloc de la condition. Deux instructions `else` dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Sachez qu'il est heureusement possible d'imbriquer des conditions et, dans ce cas, l'indentation permet de comprendre clairement le schéma d'exécution du programme. Je vous laisse essayer cette possibilité, je ne vais pas tout faire à votre place non plus :)

De nouveaux opérateurs

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**. Je vais les présenter très brièvement, vous laissant l'initiative de faire des tests car ils ne sont réellement pas difficiles à comprendre.

Opérateur	Signification littérale
$<$	Strictement inférieur à
$>$	Strictement supérieur à
$<=$	Inférieur ou égal à
$>=$	Supérieur ou égal à
$==$	Égal à
$!=$	Différent de



Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

Prédicats et booléens

Avant d'aller plus loin, sachez que les conditions qui se trouvent, par exemple, entre `if` et les deux points sont appelés des **prédicats**. Vous pouvez tester ces prédicats directement dans l'interpréteur pour comprendre les explications qui vont suivre.

```
>>> a = 0
>>> a == 5
False
>>> a > -8
True
>>> a != 33.19
True
>>>
```

L'interpréteur renvoie tantôt `True` (c'est-à-dire « vrai »), tantôt `False` (c'est-à-dire « faux »).

`True` et `False` sont les deux valeurs possibles d'un type que nous n'avons pas vu jusqu'ici : le type booléen (bool).



N'oubliez pas que True et False sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrire True sans un 'T' majuscule, Python ne va pas comprendre.

Les variables de ce type ne peuvent prendre comme valeur que vrai ou faux et peuvent être pratiques, justement, pour stocker des prédicats, de la façon que nous avons vue ou d'une façon plus détournée.

```
>>> age = 21
>>> majeur = False
>>> if age >= 18:
>>>     majeur = True
>>>
```

À la fin de cet exemple, majeur vaut `True`, c'est-à-dire « vrai », si l'âge est supérieur ou égal à 18. Sinon, il continue de valoir `False`. Les booléens ne vous semblent peut-être pas très utiles pour l'instant mais vous verrez qu'ils rendent de grands services !

Les mots-clés and, or et not

Il arrive souvent que nos conditions doivent tester plusieurs prédicats, par exemple quand l'on cherche à vérifier si une variable quelconque, de type entier, se trouve dans un intervalle précis (c'est-à-dire comprise entre deux nombres). Avec nos méthodes actuelles, le plus simple serait d'écrire :

```
# On fait un test pour savoir si a est comprise dans l'
    intervalle allant de 2 à 8 inclus
a = 5
if a >= 2:
    if a <= 8:
        print("a est dans l'intervalle.")
    else:
        print("a n'est pas dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Cela marche mais c'est assez lourd, d'autant que, pour être sûr qu'un message soit affiché à chaque fois, il faut fermer chacune des deux conditions à l'aide d'un `else` (la seconde étant imbriquée dans la première). Si vous avez du mal à comprendre cet exemple, prenez le temps de le décortiquer, ligne par ligne, il n'y a rien que de très simple.

Il existe cependant le mot clé `and` (qui signifie « et » en anglais) qui va nous rendre ici un fier service. En effet, on cherche à tester à la fois si `a` est supérieur ou égal à 2 et inférieur ou égal à 8. On peut donc réduire ainsi les conditions imbriquées :

```
if a>=2 and a<=8:
    print("a est dans l'intervalle.")
```

```
else:
    print("a n'est pas dans l'intervalle.")
```

Simple et bien plus compréhensible, avouez-le.

Sur le même mode, il existe le mot clé `or` qui signifie cette fois « ou ». Nous allons prendre le même exemple, sauf que nous allons évaluer notre condition différemment.

Nous allons chercher à savoir si `a` n'est pas dans l'intervalle. La variable ne se trouve pas dans l'intervalle si elle est inférieure à 2 ou supérieure à 8. Voici donc le code :

```
if a<2 or a>8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Enfin, il existe le mot clé `not` qui « inverse » un prédicat. Le prédicat `not a==5` équivaut donc à `a!=5`.

`not` rend la syntaxe plus claire. Pour cet exemple, j'ajoute à la liste un nouveau mot clé, `is`, qui teste l'égalité non pas des valeurs de deux variables, mais de leurs références. Je ne vais pas rentrer dans le détail de ce mécanisme avant longtemps. Il vous suffit de savoir que pour les entiers, les flottants et les booléens, c'est strictement la même chose. Mais pour tester une égalité entre variables dont le type est plus complexe, préférez l'opérateur « == ». Revenons à cette démonstration :

```
>>> majeur = False
>>> if majeur is not True:
...     print("Vous n'êtes pas encore majeur.")
...
Vous n'êtes pas encore majeur.
>>>
```

Si vous parlez un minimum l'anglais, ce prédicat est limpide et d'une simplicité sans égale.

Vous pouvez tester des prédicats plus complexes de la même façon que les précédents, en les saisissant directement, sans le `if` ni les deux points, dans l'interpréteur de commande. Vous pouvez utiliser les parenthèses ouvrantes et fermantes pour encadrer des prédicats et les comparer suivant des priorités bien précises (nous verrons ce point plus loin, si vous n'en comprenez pas l'utilité).

Votre premier programme !



À quoi on joue ?

L'heure du premier TP est venue. Comme il s'agit du tout premier, et parce qu'il y a quelques indications que je dois vous donner pour que vous parveniez jusqu'au bout, je vous accompagnerai pas à pas dans sa réalisation.

Avant de commencer

Vous allez dans cette section écrire votre premier programme. Vous allez sûrement tester les syntaxes directement dans l'interpréteur de commandes.



Vous pourriez préférer écrire votre code directement dans un fichier que vous pouvez ensuite exécuter. Si c'est le cas, je vous renvoie au chapitre traitant de ce point, que vous trouverez à la page 349 de ce livre.

Sujet

Le but de notre programme est de déterminer si une année saisie par l'utilisateur est bissextile. Il s'agit d'un sujet très prisé des enseignants en informatique quand il s'agit d'expliquer les conditions. Mille pardons, donc, à ceux qui ont déjà fait cet exercice dans un autre langage mais je trouve que ce petit programme reprend assez de thèmes abordés dans ce chapitre pour être réellement intéressant.

Je vous rappelle les règles qui déterminent si une année est bissextile ou non (vous allez peut-être même apprendre des choses que le commun des mortels ignore).

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

- Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.
- Si elle est multiple de 4, on regarde si elle est multiple de 100.
  - Si c'est le cas, on regarde si elle est multiple de 400.
    - Si c'est le cas, l'année est bissextile.
    - Sinon, elle n'est pas bissextile.
  - Sinon, elle est bissextile.

Solution ou résolution

Voilà. Le problème est posé clairement (sinon relisez attentivement l'énoncé autant de fois que nécessaire), il faut maintenant réfléchir à sa résolution en termes de programmation. C'est une phase de transition assez délicate de prime abord et je vous conseille de schématiser le problème, de prendre des notes sur les différentes étapes, sans pour l'instant penser au code. C'est une phase purement algorithmique, autrement dit, on réfléchit au programme sans réfléchir au code proprement dit.

Vous aurez besoin, pour réaliser ce petit programme, de quelques indications qui sont réellement spécifiques à Python. Ne lisez donc ceci qu'après avoir cerné et clairement écrit le problème d'une façon plus algorithmique. Cela étant dit, si vous peinez à trouver

une solution, ne vous y attardez pas. Cette phase de réflexion est assez difficile au début et, parfois il suffit d'un peu de pratique et d'explications pour comprendre l'essentiel.

La fonction `input()`

Tout d'abord, j'ai mentionné une année saisie par l'utilisateur. En effet, depuis tout à l'heure, nous testons des variables que nous déclarons nous-mêmes, avec une valeur précise. La condition est donc assez ridicule.

`input()` est une fonction qui va, pour nous, caractériser nos premières interactions avec l'utilisateur : le programme réagira différemment en fonction du nombre saisi par l'utilisateur.

`input()` accepte un paramètre facultatif : le message à afficher à l'utilisateur. Cette instruction interrompt le programme et attend que l'utilisateur saisisse ce qu'il veut puis appuie sur `Entrée`. À cet instant, la fonction renvoie ce que l'utilisateur a saisi. Il faut donc piéger cette valeur dans une variable.

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

Il subsiste un problème : le type de la variable `annee` après l'appel à `input()` est... une chaîne de caractères. Vous pouvez vous en rendre compte grâce aux apostrophes qui encadrent la valeur de la variable quand vous l'affichez directement dans l'interpréteur.

C'est bien ennuyeux : nous qui voulions travailler sur un entier, nous allons devoir convertir cette variable. Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction (c'est d'ailleurs exactement ce que c'est).

```
>>> type(annee)
<type 'str'>
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

Bon, parfait ! On a donc maintenant l'année sous sa forme entière. Notez que, si vous saisissez des lettres lors de l'appel à `input()`, la conversion renverra une erreur.



L'appel à la fonction `int()` en a peut-être déconcerté certains. On passe en paramètre de cette fonction la variable contenant la chaîne de caractères issue de `input()`, pour tenter de la convertir. La fonction `int()` renvoie la valeur convertie en entier et on la récupère donc dans la même variable. On évite ainsi de travailler sur plusieurs variables, sachant que la première n'a plus aucune utilité à présent qu'on l'a convertie.

Test de multiples

Certains pourraient également se demander comment tester si un nombre a est multiple d'un nombre b. Il suffit, en fait, de tester le reste de la division entière de b par a. Si ce reste est nul, alors a est un multiple de b.

```
>>> 5 % 2 # 5 n'est pas un multiple de 2
1
>>> 8 % 2 # 8 est un multiple de 2
0
>>>
```

À vous de jouer

Je pense vous avoir donné tous les éléments nécessaires pour réussir. À mon avis, le plus difficile est la phase de réflexion qui précède la composition du programme. Si vous avez du mal à réaliser cette opération, passez à la correction et étudiez-la soigneusement. Sinon, on se retrouve à la section suivante.

Bonne chance !

Correction

C'est l'heure de comparer nos méthodes et, avant de vous divulguer le code de ma solution, je vous précise qu'elle est loin d'être la seule possible. Vous pouvez très bien avoir trouvé quelque chose de différent mais qui fonctionne tout aussi bien.

Attention… la voiiiciiiiiii…

```
# Programme testant si une année, saisie par l'utilisateur,
# est bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre
bissextile = False # On crée un booléen qui vaut vrai ou faux
                   # selon que l'année est bissextile ou non

```

```
if annee % 400 == 0:
    bissextile = True
elif annee % 100 == 0:
    bissextile = False
elif annee % 4 == 0:
    bissextile = True
else:
    bissextile = False

if bissextile: # Si l'année est bissextile
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers afin de les exécuter. Je vous renvoie à la page 349 pour plus d'informations.

Je pense que le code est assez clair, reste à expliciter l'enchaînement des conditions. Vous remarquerez qu'on a inversé le problème. On teste en effet d'abord si l'année est un multiple de 400, ensuite si c'est un multiple de 100, et enfin si c'est un multiple de 4. En effet, le `elif` garantit que, si `annee` est un multiple de 100, ce n'est pas un multiple de 400 (car le cas a été traité au-dessus). De cette façon, on s'assure que tous les cas sont gérés. Vous pouvez faire des essais avec plusieurs années et vous rendre compte si le programme a raison ou pas.



L'utilisation de `bissextile` comme d'un prédicat à part entière vous a peut-être déconcertés. C'est en fait tout à fait possible et logique, puisque `bissextile` est un booléen. Il est de ce fait vrai ou faux et donc on peut le tester simplement. On peut bien entendu aussi écrire `if bissextile==True:`, cela revient au même.

Un peu d'optimisation

Ce qu'on a fait était bien mais on peut l'améliorer. D'ailleurs, vous vous rendrez compte que c'est presque toujours le cas. Ici, il s'agit bien entendu de notre condition, que je vais passer au crible afin d'en construire une plus courte et plus logique, si possible. On peut parler d'optimisation dans ce cas, même si l'optimisation intègre aussi et surtout les ressources consommées par votre application, en vue de diminuer ces ressources et d'améliorer la rapidité de l'application. Mais, pour une petite application comme celle-ci, je ne pense pas qu'on perdra du temps sur l'optimisation du temps d'exécution.

Le premier détail que vous auriez pu remarquer, c'est que le `else` de fin est inutile. En effet, la variable `bissextile` vaut par défaut `False` et conserve donc cette valeur si le cas n'est pas traité (ici, quand l'année n'est ni un multiple de 400, ni un multiple de 100, ni un multiple de 4).

Ensuite, il apparaît que nous pouvons faire un grand ménage dans notre condition car les deux seuls cas correspondant à une année bissextile sont « si l'année est un multiple

de 400 »ou « si l'année est un multiple de 4 mais pas de 100 ».

Le prédicat correspondant est un peu délicat, il fait appel aux priorités des parenthèses. Je ne m'attendais pas que vous le trouviez tout seuls mais je souhaite que vous le compreniez bien à présent.

```
# Programme testant si une année, saisie par l'utilisateur, est
    bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'
    utilisateur saisisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre

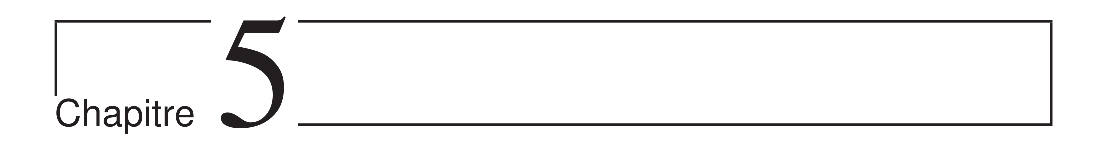
if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

Du coup, on n'a plus besoin de la variable `bissextile`, c'est déjà cela de gagné. Nous sommes passés de 16 lignes de code à seulement 7 (sans compter les commentaires et les sauts de ligne) ce qui n'est pas rien.

En résumé

- Les conditions permettent d'exécuter certaines instructions dans certains cas, d'autres instructions dans un autre cas.
- Les conditions sont marquées par les mot-clés `if` (« si »), `elif` (« sinon si ») et `else` (« sinon »).
- Les mot-clés `if` et `elif` doivent être suivis d'un test (appelé aussi prédicat).
- Les booléens sont des données soit vraies (`True`) soit fausses (`False`).



Chapitre 5

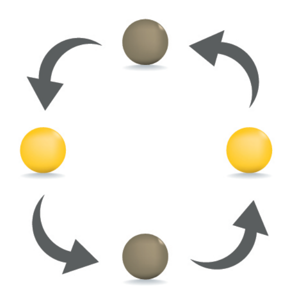
Les boucles

Difficulté : 

Les boucles sont un concept nouveau pour vous. Elles vont vous permettre de répéter une certaine opération autant de fois que nécessaire. Le concept risque de vous sembler un peu théorique car les applications pratiques présentées dans ce chapitre ne vous paraîtront probablement pas très intéressantes. Toutefois, il est impératif que cette notion soit comprise avant que vous ne passiez à la suite. Viendra vite le moment où vous aurez du mal à écrire une application sans boucle.

En outre, les boucles peuvent permettre de parcourir certaines séquences comme les chaînes de caractères pour, par exemple, en extraire chaque caractère.

Alors, on commence ?



En quoi cela consiste-t-il ?

Comme je l'ai dit juste au-dessus, les boucles constituent un moyen de répéter un certain nombre de fois des instructions de votre programme. Prenons un exemple simple, même s'il est assez peu réjouissant en lui-même : écrire un programme affichant la table de multiplication par 7, de 1 * 7 à 10 * 7.

... bah quoi ?

Bon, ce n'est qu'un exemple, ne faites pas cette tête, et puis je suis sûr que ce sera utile pour certains. Dans un premier temps, vous devriez arriver au programme suivant :

```
print(" 1 * 7 =", 1 * 7)
print(" 2 * 7 =", 2 * 7)
print(" 3 * 7 =", 3 * 7)
print(" 4 * 7 =", 4 * 7)
print(" 5 * 7 =", 5 * 7)
print(" 6 * 7 =", 6 * 7)
print(" 7 * 7 =", 7 * 7)
print(" 8 * 7 =", 8 * 7)
print(" 9 * 7 =", 9 * 7)
print("10 * 7 =", 10 * 7)
```

... et le résultat :

```
 1 * 7 = 7
 2 * 7 = 14
 3 * 7 = 21
 4 * 7 = 28
 5 * 7 = 35
 6 * 7 = 42
 7 * 7 = 49
 8 * 7 = 56
 9 * 7 = 63
10 * 7 = 70
```



Je vous rappelle que vous pouvez enregistrer vos codes dans des fichiers. Vous trouverez la marche à suivre à la page 349 de ce livre.

Bon, c'est sûrement la première idée qui vous est venue et cela fonctionne, très bien même. Seulement, vous reconnaîtrez qu'un programme comme cela n'est pas bien utile. Essayons donc le même programme mais, cette fois-ci, en utilisant une variable ; ainsi, si on décide d'afficher la table de multiplication de 6, on n'aura qu'à changer la valeur de la variable ! Pour cet exemple, on utilise une variable `nb` qui contiendra 7. Les instructions seront légèrement différentes mais vous devriez toujours pouvoir écrire ce programme :

```
nb = 7
print(" 1 *", nb, "=", 1 * nb)
```

```
print(" 2 *", nb, "=", 2 * nb)
print(" 3 *", nb, "=", 3 * nb)
print(" 4 *", nb, "=", 4 * nb)
print(" 5 *", nb, "=", 5 * nb)
print(" 6 *", nb, "=", 6 * nb)
print(" 7 *", nb, "=", 7 * nb)
print(" 8 *", nb, "=", 8 * nb)
print(" 9 *", nb, "=", 9 * nb)
print("10 *", nb, "=", 10 * nb)
```

Le résultat est le même, vous pouvez vérifier. Mais le code est quand-même un peu plus intéressant : on peut changer la table de multiplication à afficher en changeant la valeur de la variable `nb`.

Mais ce programme reste assez peu pratique et il accomplit une tâche bien répétitive. Les programmeurs étant très paresseux, ils préfèrent utiliser les boucles.

La boucle `while`

La boucle que je vais présenter se retrouve dans la plupart des autres langages de programmation et porte le même nom. Elle permet de répéter un **bloc d'instructions** tant qu'une condition est vraie (`while` signifie « tant que » en anglais). J'espère que le concept de **bloc d'instructions** est clair pour vous, sinon je vous renvoie au chapitre précédent.

La syntaxe de `while` est :

```
while condition:
    # instruction 1
    # instruction 2
    # ...
    # instruction N
```

Vous devriez reconnaître la forme d'un bloc d'instructions, du moins je l'espère.



Quelle condition va-t-on utiliser ?

Eh bien, c'est là le point important. Dans cet exemple, on va créer une variable qui sera incrémentée dans le bloc d'instructions. Tant que cette variable sera inférieure à 10, le bloc s'exécutera pour afficher la table.

Si ce n'est pas clair, regardez ce code, quelques commentaires suffiront pour le comprendre :

```
nb = 7 # On garde la variable contenant le nombre dont on veut la table de multiplication
i = 0 # C'est notre variable compteur que nous allons incrémenter dans la boucle
```

```
3
4 while i < 10: # Tant que i est strictement inférieure à 10
5     print(i + 1, "*", nb, "=", (i + 1) * nb)
6     i += 1 # On incrémente i de 1 à chaque tour de boucle
```

Analysons ce code ligne par ligne :

1. On instancie la variable `nb` qui accueille le nombre sur lequel nous allons travailler (en l'occurence, 7). Vous pouvez bien entendu faire saisir ce nombre par l'utilisateur, vous savez le faire à présent.
2. On instancie la variable `i` qui sera notre compteur durant la boucle. `i` est un standard utilisé quand il est question de boucles et de variables s'incrémentant mais il va de soi que vous auriez pu lui donner un autre nom. On l'initialise à 0.
3. Un saut de ligne ne fait jamais de mal !
4. On trouve ici l'instruction `while` qui se décode, comme je l'ai indiqué en commentaire, en « **tant que i est strictement inférieure à 10** ». N'oubliez pas les deux points à la fin de la ligne.
5. La ligne du `print`, vous devez la reconnaître. Maintenant, la plus grande partie de la ligne affichée est constituée de variables, à part les signes mathématiques. Vous remarquez qu'à chaque fois qu'on utilise `i` dans cette ligne, pour l'affichage ou le calcul, on lui ajoute 1 : cela est dû au fait qu'en programmation, on a l'habitude (habitude que vous devrez prendre) de commencer à compter à partir de 0. Seulement ce n'est pas le cas de la table de multiplication, qui va de 1 à 10 et non de 0 à 9, comme c'est le cas pour les valeurs de `i`. Certes, j'aurais pu changer la condition et la valeur initiale de `i`, ou même placer l'incrémentation de `i` avant l'affichage, mais j'ai voulu prendre le cas le plus courant, le format de boucle que vous retrouverez le plus souvent. Rien ne vous empêche de faire les tests et je vous y encourage même.
6. Ici, on incrémente la variable `i` de 1. Si on est dans le premier tour de boucle, `i` passe donc de 0 à 1. Et alors, puisqu'il s'agit de la fin du bloc d'instructions, on revient à l'instruction `while`. `while` vérifie que la valeur de `i` est toujours inférieure à 10. Si c'est le cas (et ça l'est pour l'instant), on exécute à nouveau le bloc d'instructions. En tout, on exécute ce bloc 10 fois, jusqu'à ce que `i` passe de 9 à 10. Alors, l'instruction `while` vérifie la condition, se rend compte qu'elle est à présent fausse (la valeur de `i` n'est pas inférieure à 10 puisqu'elle est maintenant égale à 10) et s'arrête. S'il y avait du code après le bloc, il serait à présent exécuté.



N'oubliez pas d'incrémenter `i` ! Sinon, vous créez ce qu'on appelle une boucle infinie, puisque la valeur de `i` n'est jamais supérieure à 10 et la condition du `while`, par conséquent, toujours vraie... La boucle s'exécute donc à l'infini, du moins en théorie. Si votre ordinateur se lance dans une boucle infinie à cause de votre programme, pour interrompre la boucle, vous devrez taper CTRL + C dans la fenêtre de l'interpréteur (sous Windows ou Linux). Python ne le fera pas tout seul car, pour lui, il se passe bel et bien quelque chose. De toute façon, il est incapable de différencier une boucle infinie d'une boucle finie : c'est au programmeur de le faire.

La boucle `for`

Comme je l'ai dit précédemment, on retrouve l'instruction `while` dans la plupart des autres langages. Dans le C++ ou le Java, on retrouve également des instructions `for` mais qui n'ont pas le même sens. C'est assez particulier et c'est le point sur lequel je risque de manquer d'exemples dans l'immédiat, toute son utilité se révélant au chapitre sur les listes. Notez que, si vous avez fait du Perl ou du PHP, vous pouvez retrouver les boucles `for` sous un mot-clé assez proche : `foreach`.

L'instruction `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Nous n'avons pas vu (et nous ne verrons pas tout de suite) ces séquences assez particulières mais très répandues, même si elles peuvent se révéler complexes. Toutefois, il en existe un type que nous avons rencontré depuis quelque temps déjà : les chaînes de caractères.

Les chaînes de caractères sont des séquences... de caractères ! Vous pouvez parcourir une chaîne de caractères (ce qui est également possible avec `while` mais nous verrons plus tard comment). Pour l'instant, intéressons-nous à `for`.

L'instruction `for` se construit ainsi :

```
for element in sequence:
```

`element` est une variable créée par le `for`, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.

Ce n'est pas très clair ? Alors, comme d'habitude, tout s'éclaire avec le code !

```
chaine = "Bonjour les ZEROS"
for lettre in chaine:
    print(lettre)
```

Ce qui nous donne le résultat suivant :

```
B
o
n
j
o
u
r

l
e
s

Z
E
R
O
S
```

Est-ce plus clair ? En fait, la variable `lettre` prend successivement la valeur de chaque lettre contenue dans la chaîne de caractères (d'abord B, puis o, puis n...). On affiche ces valeurs avec `print` et cette fonction revient à la ligne après chaque message, ce qui fait que toutes les lettres sont sur une seule colonne. Littéralement, la ligne 2 signifie « **pour lettre dans chaine** ». Arrivé à cette ligne, l'interpréteur va créer une variable `lettre` qui contiendra le premier élément de la chaîne (autrement dit, la première lettre). Après l'exécution du bloc, la variable `lettre` contient la seconde lettre, et ainsi de suite tant qu'il y a une lettre dans la chaîne.

Notez bien que, du coup, il est inutile d'incrémenter la variable `lettre` (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction `for`.

À l'instar des conditions que nous avons vues jusqu'ici, `in` peut être utilisée ailleurs que dans une boucle `for`.

```
1 chaine = "Bonjour les ZEROS"
2 for lettre in chaine:
3     if lettre in "AEIOUYaeiouy": # lettre est une voyelle
4         print(lettre)
5     else: # lettre est une consonne... ou plus exactement, lettre n'est pas une voyelle
6         print("*")
```

... ce qui donne :

```
*
o
*
*
o
u
*
*
*
e
*
*
*
E
*
*
*
```

Voilà ! L'interpréteur affiche les lettres si ce sont des voyelles et, sinon, il affiche des « * ». Notez bien que le 0 n'est pas affiché à la fin, Python ne se doute nullement qu'il s'agit d'un « o » stylisé.

Retenez bien cette utilisation de `in` dans une condition. On cherche à savoir si un élément quelconque est contenu dans un ensemble donné (ici, si la lettre est contenue dans « AEIOUYaeiouy », c'est-à-dire si `lettre` est une voyelle). On retrouvera plus

loin cette fonctionnalité.

Un petit bonus : les mots-clés `break` et `continue`

Je vais ici vous montrer deux nouveaux mots-clés, `break` et `continue`. Vous ne les utiliserez peut-être pas beaucoup mais vous devez au moins savoir qu'ils existent… et à quoi ils servent.

Le mot-clé `break`

Le mot-clé `break` permet tout simplement d'interrompre une boucle. Il est souvent utilisé dans une forme de boucle que je n'approuve pas trop :

```python
while 1: # 1 est toujours vrai -> boucle infinie
    lettre = input("Tapez 'Q' pour quitter : ")
    if lettre == "Q":
        print("Fin de la boucle")
        break
```

La boucle `while` a pour condition `1`, c'est-à-dire une condition qui sera *toujours* vraie. Autrement dit, en regardant la ligne du `while`, on pense à une boucle infinie. En pratique, on demande à l'utilisateur de taper une lettre (un 'Q' pour quitter). Tant que l'utilisateur ne saisit pas cette lettre, le programme lui redemande de taper une lettre. Quand il tape 'Q', le programme affiche `Fin de la boucle` et la boucle s'arrête grâce au mot-clé `break`.

Ce mot-clé permet d'arrêter une boucle quelle que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

C'est un exemple un peu simpliste mais vous pouvez voir l'idée d'ensemble. Dans ce cas-là et, à mon sens, dans la plupart des cas où `break` est utilisé, on pourrait s'en sortir en précisant une véritable condition à la ligne du `while`. Par exemple, pourquoi ne pas créer un booléen qui sera **vrai** tout au long de la boucle et **faux** quand la boucle doit s'arrêter ? Ou bien tester directement si `lettre != « Q »` dans le `while` ?

Parfois, `break` est véritablement utile et fait gagner du temps. Mais ne l'utilisez pas à outrance, préférez une boucle avec une condition claire plutôt qu'un bloc d'instructions avec un `break`, qui sera plus dur à appréhender d'un seul coup d'œil.

Le mot-clé `continue`

Le mot-clé `continue` permet de… continuer une boucle, en repartant directement à la ligne du `while` ou `for`. Un petit exemple s'impose, je pense :

```python
i = 1
while i < 20: # Tant que i est inférieure à 20
    if i % 3 == 0:
        i += 4 # On ajoute 4 à i
```

```
5         print("On incrémente i de 4. i est maintenant égale à",
              i)
6         continue # On retourne au while sans exécuter les
              autres lignes
7     print("La variable i =", i)
8     i += 1 # Dans le cas classique on ajoute juste 1 à i
```

Voici le résultat :

```
La variable i = 1
La variable i = 2
On incrémente i de 4. i est maintenant égale à 7
La variable i = 7
La variable i = 8
On incrémente i de 4. i est maintenant égale à 13
La variable i = 13
La variable i = 14
On incrémente i de 4. i est maintenant égale à 19
La variable i = 19
```

Comme vous le voyez, tous les trois tours de boucle, `i` s'incrémente de 4. Arrivé au mot-clé `continue`, Python n'exécute pas la fin du bloc mais revient au début de la boucle en testant à nouveau la condition du `while`. Autrement dit, quand Python arrive à la ligne 6, il saute à la ligne 2 sans exécuter les lignes 7 et 8. Au nouveau tour de boucle, Python reprend l'exécution normale de la boucle (`continue` n'ignore la fin du bloc que pour le tour de boucle courant).

Mon exemple ne démontre pas de manière éclatante l'utilité de `continue`. Les rares fois où j'utilise ce mot-clé, c'est par exemple pour supprimer des éléments d'une liste, mais nous n'avons pas encore vu les listes. L'essentiel, pour l'instant, c'est que vous vous souveniez de ces deux mots-clés et que vous sachiez ce qu'ils font, si vous les rencontrez au détour d'une instruction. Personnellement, je n'utilise pas très souvent ces mots-clés mais c'est aussi une question de goût.

En résumé

- Une boucle sert à répéter une portion de code en fonction d'un prédicat.
- On peut créer une boucle grâce au mot-clé `while` suivi d'un prédicat.
- On peut parcourir une séquence grâce à la syntaxe `for element in sequence:`.



Chapitre 6

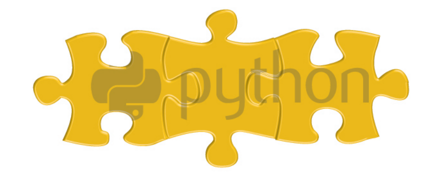
Pas à pas vers la modularité (1/2)

Difficulté : 

En programmation, on est souvent amené à utiliser plusieurs fois des groupes d'instructions dans un but très précis. Attention, je ne parle pas ici de boucles. Simplement, vous pourrez vous rendre compte que la plupart de nos tests pourront être regroupés dans des blocs plus vastes, fonctions ou modules. Je vais détailler tranquillement ces deux concepts.

Les fonctions permettent de regrouper plusieurs instructions dans un bloc qui sera appelé grâce à un nom. D'ailleurs, vous avez déjà vu des fonctions : `print` et `input` en font partie par exemple.

Les modules permettent de regrouper plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié aux mathématiques.



Les fonctions : à vous de jouer

Nous avons utilisé pas mal de fonctions depuis le début de ce tutoriel. On citera pour mémoire `print`, `type` et `input`, sans compter quelques autres. Mais vous devez bien vous rendre compte qu'il existe un nombre incalculable de fonctions déjà construites en Python. Toutefois, vous vous apercevrez aussi que, très souvent, un programmeur crée ses propres fonctions. C'est le premier pas que vous ferez, dans ce chapitre, vers la **modularité**. Ce terme un peu barbare signifie que nous allons nous habituer à regrouper dans des fonctions des parties de notre code que nous serons amenés à réutiliser. Au prochain chapitre, nous apprendrons à regrouper nos fonctions ayant un rapport entre elles dans un fichier, pour constituer un module, mais n'anticipons pas.

La création de fonctions

Nous allons, pour illustrer cet exemple, reprendre le code de la table de multiplication, que nous avons vu au chapitre précédent et qui, décidément, n'en finit pas de vous poursuivre.

Nous allons emprisonner notre code calculant la table de multiplication par 7 dans une fonction que nous appellerons `table_par_7`.

On crée une fonction selon le schéma suivant :

```
def nom_de_la_fonction(parametre1, parametre2, parametre3,
    parametreN):
    # Bloc d'instructions
```

Les blocs d'instructions nous courent après aussi, quel enfer. Si l'on décortique la ligne de définition de la fonction, on trouve dans l'ordre :

- `def`, mot-clé qui est l'abréviation de « define » (définir, en anglais) et qui constitue le prélude à toute construction de fonction.
- Le nom de la fonction, qui se nomme exactement comme une variable (nous verrons par la suite que ce n'est pas par hasard). N'utilisez pas un nom de variable déjà instanciée pour nommer une fonction.
- La liste des paramètres qui seront fournis lors d'un appel à la fonction. Les paramètres sont séparés par des virgules et la liste est encadrée par des parenthèses ouvrante et fermante (là encore, les espaces sont optionnels mais améliorent la lisibilité).
- Les deux points, encore et toujours, qui clôturent la ligne.



Les parenthèses sont obligatoires, quand bien même votre fonction n'attendrait aucun paramètre.

Le code pour mettre notre table de multiplication par 7 dans une fonction serait donc :

```
def table_par_7():
    nb = 7
    i = 0 # Notre compteur ! L'auriez-vous oublié ?
```

```
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Quand vous exécutez ce code à l'écran, il ne se passe rien. Une fois que vous avez retrouvé les trois chevrons, essayez d'appeler la fonction :

```
>>> table_par_7()
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
>>>
```

Bien, c'est, euh, exactement ce qu'on avait réussi à faire au chapitre précédent et l'intérêt ne saute pas encore aux yeux. L'avantage est que l'on peut appeler facilement la fonction et réafficher toute la table sans avoir besoin de tout réécrire !



Mais, si on saisit des paramètres pour pouvoir afficher la table de 5 ou de 8. . . ?

Oui, ce serait déjà bien plus utile. Je ne pense pas que vous ayez trop de mal à trouver le code de la fonction :

```
def table(nb):
    i = 0
    while i < 10: # Tant que i est strictement inférieure à 10,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1 # On incrémente i de 1 à chaque tour de boucle.
```

Et là, vous pouvez passer en argument différents nombres, `table(8)` pour afficher la table de multiplication par 8 par exemple.

On peut aussi envisager de passer en paramètre le nombre de valeurs à afficher dans la table.

```
def table(nb, max):
    i = 0
    while i < max: # Tant que i est strictement inférieure à la
        variable max,
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Si vous tapez à présent `table(11, 20)`, l'interpréteur vous affichera la table de 11, de 1*11 à 20*11. Magique non ?



Dans le cas où l'on utilise plusieurs paramètres sans les nommer, comme ici, il faut respecter l'ordre d'appel des paramètres, cela va de soi. Si vous commencez à mettre le nombre d'affichages en premier paramètre alors que, dans la définition, c'était le second, vous risquez d'avoir quelques surprises. Il est possible d'appeler les paramètres dans le désordre mais il faut, dans ce cas, préciser leur nom : nous verrons cela plus loin.

Si vous fournissez en second paramètre un nombre négatif, vous avez toutes les chances de créer une magnifique boucle infinie… vous pouvez l'empêcher en rajoutant des vérifications avant la boucle : par exemple, si le nombre est négatif ou nul, je le mets à 10. En Python, on préférera mettre un commentaire en tête de fonction ou une `docstring`, comme on le verra ultérieurement, pour indiquer que `max` doit être positif, plutôt que de faire des vérifications qui au final feront perdre du temps. Une des phrases reflétant la philosophie du langage et qui peut s'appliquer à ce type de situation est « *we're all consenting adults here* »[1] (sous-entendu, quelques avertissements en commentaires sont plus efficaces qu'une restriction au niveau du code). On aura l'occasion de retrouver cette phrase plus loin, surtout quand on parlera des objets.

Valeurs par défaut des paramètres

On peut également préciser une valeur par défaut pour les paramètres de la fonction. Vous pouvez par exemple indiquer que le nombre maximum d'affichages doit être de 10 par défaut (c'est-à-dire si l'utilisateur de votre fonction ne le précise pas). Cela se fait le plus simplement du monde :

```
def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb
    de 1*nb à max*nb

    (max >= 0)"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Il suffit de rajouter `=10` après `max`. À présent, vous pouvez appeler la fonction de deux façons : soit en précisant le numéro de la table et le nombre maximum d'affichages, soit en ne précisant que le numéro de la table (`table(7)`). Dans ce dernier cas, `max` vaudra 10 par défaut.

J'en ai profité pour ajouter quelques lignes d'explications que vous aurez sans doute remarquées. Nous avons placé une chaîne de caractères, sans la capturer dans une variable, juste en-dessous de la définition de la fonction. Cette chaîne est ce qu'on

1. « Nous sommes entre adultes consentants ».

appelle une `docstring` que l'on pourrait traduire par une chaîne d'aide. Si vous tapez `help(table)`, c'est ce message que vous verrez apparaître. Documenter vos fonctions est également une bonne habitude à prendre. Comme vous le voyez, on indente cette chaîne et on la met entre triple guillemets. Si la chaîne figure sur une seule ligne, on pourra mettre les trois guillemets fermants sur la même ligne ; sinon, on préférera sauter une ligne avant de fermer cette chaîne, pour des raisons de lisibilité. Tout le texte d'aide est indenté au même niveau que le code de la fonction.

Enfin, sachez que l'on peut appeler des paramètres par leur nom. Cela est utile pour une fonction comptant un certain nombre de paramètres qui ont tous une valeur par défaut. Vous pouvez aussi utiliser cette méthode sur une fonction sans paramètre par défaut, mais c'est moins courant.

Prenons un exemple de définition de fonction :

```
def fonc(a=1, b=2, c=3, d=4, e=5):
    print("a =", a, "b =", b, "c =", c, "d =", d, "e =", e)
```

Simple, n'est-ce pas ? Eh bien, vous avez de nombreuses façons d'appeler cette fonction. En voici quelques exemples :

Instruction	**Résultat**
`fonc()`	a = 1 b = 2 c = 3 d = 4 e = 5
`fonc(4)`	a = 4 b = 2 c = 3 d = 4 e = 5
`fonc(b=8, d=5)`	a = 1 b = 8 c = 3 d = 5 e = 5
`fonc(b=35, c=48, a=4, e=9)`	a = 4 b = 35 c = 48 d = 4 e = 9

Je ne pense pas que des explications supplémentaires s'imposent. Si vous voulez changer la valeur d'un paramètre, vous tapez son nom, suivi d'un signe égal puis d'une valeur (qui peut être une variable bien entendu). Peu importent les paramètres que vous précisez (comme vous le voyez dans cet exemple où tous les paramètres ont une valeur par défaut, vous pouvez appeler la fonction sans paramètre), peu importe l'ordre d'appel des paramètres.

Signature d'une fonction

On entend par « signature de fonction » les éléments qui permettent au langage d'identifier ladite fonction. En C++, par exemple, la signature d'une fonction est constituée de son nom et du type de chacun de ses paramètres. Cela veut dire que l'on peut trouver plusieurs fonctions portant le même nom mais dont les paramètres diffèrent. Au moment de l'appel de fonction, le compilateur recherche la fonction qui s'applique à cette signature.

En Python comme vous avez pu le voir, on ne précise pas les types des paramètres. Dans ce langage, la signature d'une fonction est tout simplement son nom. Cela signifie que vous ne pouvez définir deux fonctions du même nom (si vous le faites, l'ancienne définition est écrasée par la nouvelle).

```
def exemple():
```

```
2       print("Un exemple d'une fonction sans paramètre")
3
4   exemple()
5
6   def exemple(): # On redéfinit la fonction exemple
7       print("Un autre exemple de fonction sans paramètre")
8
9   exemple()
```

A la ligne 1 on définit la fonction `exemple`. On l'appelle une première fois à la ligne 4. On redéfinit à la ligne 6 la fonction `exemple`. L'ancienne définition est écrasée et l'ancienne fonction ne pourra plus être appelée.

Retenez simplement que, comme pour les variables, un nom de fonction ne renvoie que vers une fonction unique, on ne peut surcharger de fonctions en Python.

L'instruction `return`

Ce que nous avons fait était intéressant, mais nous n'avons pas encore fait le tour des possibilités de la fonction. Et d'ailleurs, même à la fin de ce chapitre, il nous restera quelques petites fonctionnalités à voir. Si vous vous souvenez bien, il existe des fonctions comme `print` qui ne renvoient rien (attention, « renvoyer » et « afficher » sont deux choses différentes) et des fonctions telles que `input` ou `type` qui renvoient une valeur. Vous pouvez capturer cette valeur en plaçant une variable devant (exemple `variable2 = type(variable1)`). En effet, les fonctions travaillent en général sur des données et renvoient le résultat obtenu, suite à un calcul par exemple.

Prenons un exemple simple : une fonction chargée de mettre au carré une valeur passée en argument. Je vous signale au passage que Python en est parfaitement capable sans avoir à coder une nouvelle fonction, mais c'est pour l'exemple.

```
def carre(valeur):
    return valeur * valeur
```

L'instruction `return` signifie qu'on va **renvoyer**[2] la valeur, pour pouvoir la récupérer ensuite et la stocker dans une variable par exemple. Cette instruction arrête le déroulement de la fonction, le code situé après le `return` ne s'exécutera pas.

```
variable = carre(5)
```

La variable `variable` contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Sachez que l'on peut renvoyer plusieurs valeurs que l'on sépare par des virgules, et que l'on peut les capturer dans des variables également séparées par des virgules, mais je m'attarderai plus loin sur cette particularité. Retenez simplement la définition d'une fonction, les paramètres, les valeurs par défaut, l'instruction `return` et ce sera déjà bien.

2. Certains d'entre vous ont peut-être l'habitude d'employer le mot « retourner » ; il s'agit d'un anglicisme et je lui préfère l'expression « renvoyer ».

Les fonctions `lambda`

Nous venons de voir comment créer une fonction grâce au mot-clé `def`. Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes car limitées à une seule instruction.



Pourquoi une autre façon de créer des fonctions ? La première suffit, non ?

Disons que ce n'est pas tout à fait la même chose, comme vous allez le voir. Les fonctions lambda sont en général utilisées dans un certain contexte, pour lequel définir une fonction à l'aide de `def` serait plus long et moins pratique.

Syntaxe

Avant tout, voyons la syntaxe d'une définition de fonction `lambda`. Nous allons utiliser le mot-clé `lambda` comme ceci : `lambda arg1, arg2,... : instruction de retour`.

Je pense qu'un exemple vous semblera plus clair. On veut créer une fonction qui prend un paramètre et renvoie ce paramètre au carré.

```
>>> lambda x: x * x
<function <lambda> at 0x00BA1B70>
>>>
```

D'abord, on a le mot-clé `lambda` suivi de la liste des arguments, séparés par des virgules. Ici, il n'y a qu'un seul argument, c'est `x`. Ensuite figure un nouveau signe deux points « : » et l'instruction de la `lambda`. C'est le résultat de l'instruction que vous placez ici qui sera renvoyé par la fonction. Dans notre exemple, on renvoie donc `x * x`.



Comment fait-on pour appeler notre `lambda` ?

On a bien créé une fonction `lambda` mais on ne dispose ici d'aucun moyen pour l'appeler. Vous pouvez tout simplement stocker votre fonction `lambda` nouvellement définie dans une variable, par une simple affectation :

```
>>> f = lambda x: x * x
>>> f(5)
25
>>> f(-18)
324
>>>
```

Un autre exemple : si vous voulez créer une fonction `lambda` prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

```
lambda x, y: x + y
```

Utilisation

À notre niveau, les fonctions `lambda` sont plus une curiosité que véritablement utiles. Je vous les présente maintenant parce que le contexte s'y prête et que vous pourriez en rencontrer certaines sans comprendre ce que c'est.

Il vous faudra cependant attendre un peu pour que je vous montre une réelle application des `lambda`. En attendant, n'oubliez pas ce mot-clé et la syntaxe qui va avec… on passe à la suite !

À la découverte des modules

Jusqu'ici, nous avons travaillé avec les fonctions de Python chargées au lancement de l'interpréteur. Il y en a déjà un certain nombre et nous pourrions continuer et finir cette première partie sans utiliser de module Python… ou presque. Mais il faut bien qu'à un moment, je vous montre cette possibilité des plus intéressantes !

Les modules, qu'est-ce que c'est ?

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier. On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à **importer** le module et utiliser ensuite toutes les fonctions et variables prévues.

Il existe un grand nombre de modules disponibles avec Python sans qu'il soit nécessaire d'installer des bibliothèques supplémentaires. Pour cette partie, nous prendrons l'exemple du module `math` qui contient, comme son nom l'indique, des fonctions mathématiques. Inutile de vous inquiéter, nous n'allons pas nous attarder sur le module lui-même pour coder une calculatrice scientifique, nous verrons surtout les différentes méthodes d'importation.

La méthode `import`

Lorsque vous ouvrez l'interpréteur Python, les fonctionnalités du module `math` ne sont pas incluses. Il s'agit en effet d'un module, il vous appartient de l'importer si vous vous dites « tiens, mon programme risque d'avoir besoin de fonctions mathématiques ». Nous allons voir une première syntaxe d'importation.

```
>>> import math
```

```
>>>
```

La syntaxe est facile à retenir : le mot-clé `import`, qui signifie « importer » en anglais, suivi du nom du module, ici `math`.

Après l'exécution de cette instruction, rien ne se passe... en apparence. En réalité, Python vient d'importer le module `math`. Toutes les fonctions mathématiques contenues dans ce module sont maintenant accessibles. Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :

```
>>> math.sqrt(16)
4
>>>
```

Comme vous le voyez, la fonction `sqrt` du module `math` renvoie la racine carrée du nombre passé en paramètre.



Mais comment suis-je censé savoir quelles fonctions existent et ce que fait `math.sqrt` dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt car, oui, c'est une fonction qui va nous donner la solution. Il s'agit de `help`, qui prend en argument la fonction ou le module sur lequel vous demandez de l'aide. L'aide est fournie en anglais mais c'est de l'anglais technique, c'est-à-dire une forme de l'anglais que vous devrez maîtriser pour programmer, si ce n'est pas déjà le cas. Une grande majorité de la documentation est en anglais, bien que vous puissiez maintenant en trouver une bonne part en français.

```
>>> help("math")
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
```

```
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians)
            of x.

    asin(...)
-- Suite --
```

Si vous parlez un minimum l'anglais, vous avez accès à une description exhaustive des fonctions du module `math`. Vous voyez en haut de la page le nom du module, le fichier qui l'héberge, puis la description du module. Ensuite se trouve une liste des fonctions, chacune étant accompagnée d'une courte description.

Tapez `Q` pour revenir à la fenêtre d'interpréteur, `Espace` pour avancer d'une page, `Entrée` pour avancer d'une ligne. Vous pouvez également passer un nom de fonction en paramètre de la fonction `help`.

```
>>> help("math.sqrt")
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.

>>>
```



Ne mettez pas les parenthèses habituelles après le nom de la fonction. C'est en réalité la référence de la fonction que vous envoyez à `help`. Si vous rajoutez les parenthèses ouvrantes et fermantes après le nom de la fonction, vous devrez préciser une valeur. Dans ce cas, c'est la valeur renvoyée par `math.sqrt` qui sera analysée, soit un nombre (entier ou flottant).

Nous reviendrons plus tard sur le concept des références des fonctions. Si vous avez compris pourquoi il ne fallait pas mettre de parenthèses après le nom de la fonction dans `help`, tant mieux. Sinon, ce n'est pas grave, nous y reviendrons en temps voulu.

Utiliser un espace de noms spécifique

En vérité, quand vous tapez `import math`, cela crée un espace de noms dénommé « math », contenant les variables et fonctions du module `math`. Quand vous tapez `math.sqrt(25)`, vous précisez à Python que vous souhaitez exécuter la fonction `sqrt` contenue dans l'espace de noms `math`. Cela signifie que vous pouvez avoir, dans l'espace de noms principal, une autre fonction `sqrt` que vous avez définie vous-mêmes. Il n'y aura pas de conflit entre, d'une part, la fonction que vous avez créée et que vous appellerez grâce à l'instruction `sqrt` et, d'autre part, la fonction `sqrt` du module `math`

que vous appellerez grâce à l'instruction `math.sqrt`.



Mais, concrètement, un espace de noms, c'est quoi ?

Il s'agit de regrouper certaines fonctions et variables sous un préfixe spécifique. Prenons un exemple concret :

```
import math
a = 5
b = 33.2
```

Dans l'espace de noms principal, celui qui ne nécessite pas de préfixe et que vous utilisez depuis le début de ce tutoriel, on trouve :

- La variable `a`.
- La variable `b`.
- Le module `math`, qui se trouve dans un espace de noms s'appelant `math` également. Dans cet espace de noms, on trouve :
  - la fonction `sqrt` ;
  - la variable `pi` ;
  - et bien d'autres fonctions et variables...

C'est aussi l'intérêt des modules : des variables et fonctions sont stockées à part, bien à l'abri dans un espace de noms, sans risque de conflit avec vos propres variables et fonctions. Mais dans certains cas, vous pourrez vouloir changer le nom de l'espace de noms dans lequel sera stocké le module importé.

```
import math as mathematiques
mathematiques.sqrt(25)
```



Qu'est-ce qu'on a fait là ?

On a simplement importé le module `math` en spécifiant à Python de l'héberger dans l'espace de noms dénommé « mathematiques » au lieu de `math`. Cela permet de mieux contrôler les espaces de noms des modules que vous importerez. Dans la plupart des cas, vous n'utiliserez pas cette fonctionnalité mais, au moins, vous savez qu'elle existe. Quand on se penchera sur les packages, vous vous souviendrez probablement de cette possibilité.

Une autre méthode d'importation : `from ... import ...`

Il existe une autre méthode d'importation qui ne fonctionne pas tout à fait de la même façon. En fonction du résultat attendu, j'utilise indifféremment l'une ou l'autre de ces méthodes. Reprenons notre exemple du module `math`. Admettons que nous ayons

uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

```
>>> from math import fabs
>>> fabs(-5)
5
>>> fabs(2)
2
>>>
```

Pour ceux qui n'ont pas encore étudié les valeurs absolues, il s'agit tout simplement de l'opposé de la variable si elle est négative, et de la variable elle-même si elle est positive. Une valeur absolue est ainsi toujours positive.

Vous aurez remarqué qu'on ne met plus le préfixe `math.` devant le nom de la fonction. En effet, nous l'avons importée avec la méthode `from` : celle-ci charge la fonction depuis le module indiqué et la place dans l'interpréteur au même plan que les fonctions existantes, comme `print` par exemple. Si vous avez compris les explications sur les espaces de noms, vous voyez que `print` et `fabs` sont dans le même espace de noms (principal).

Vous pouvez appeler toutes les variables et fonctions d'un module en tapant « * » à la place du nom de la fonction à importer.

```
1 >>> from math import *
2 >>> sqrt(4)
3 2
4 >>> fabs(5)
5 5
```

À la ligne 1 de notre programme, l'interpréteur a parcouru toutes les fonctions et variables du module `math` et les a importées directement dans l'espace de noms principal sans les emprisonner dans l'espace de noms `math`.

Bilan



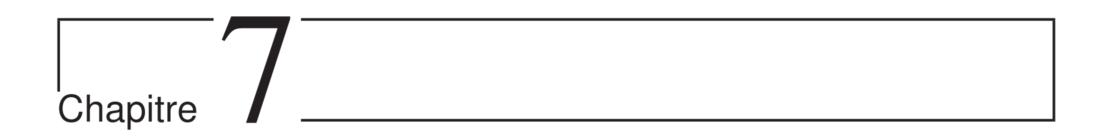
Quelle méthode faut-il utiliser ?

Vaste question ! Je dirais que c'est à vous de voir. La seconde méthode a l'avantage inestimable d'économiser la saisie systématique du nom du module en préfixe de chaque fonction. L'inconvénient de cette méthode apparaît si l'on utilise plusieurs modules de cette manière : si par hasard il existe dans deux modules différents deux fonctions portant le même nom, l'interpréteur ne conservera que la dernière fonction appelée [3]. Conclusion... c'est à vous de voir en fonction de vos besoins !

3. Je vous rappelle qu'il ne peut y avoir deux variables ou fonctions portant le même nom.

En résumé

- Une fonction est une portion de code contenant des instructions, que l'on va pouvoir réutiliser facilement.
- Découper son programme en fonctions permet une meilleure organisation.
- Les fonctions peuvent recevoir des informations en entrée et renvoyer une information grâce au mot-clé `return`.
- Les fonctions se définissent de la façon suivante : `def nom_fonction(parametre1, parametre2, parametreN):`



Chapitre 7

Pas à pas vers la modularité (2/2)

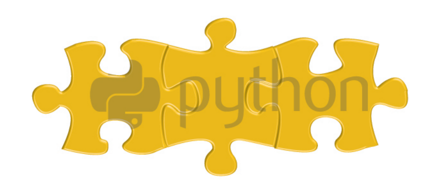
Difficulté : 

Nous allons commencer par voir comment mettre nos programmes en boîte... ou plutôt en fichier. Je vais faire d'une pierre deux coups : d'abord, c'est chouette d'avoir son programme dans un fichier modifiable à souhait, surtout qu'on commence à pouvoir faire des programmes assez sympas (même si vous n'en avez peut-être pas l'impression). Ensuite, c'est un prélude nécessaire à la création de modules.

Comme vous allez le voir, nos programmes Python peuvent être mis dans des fichiers pour être exécutés ultérieurement. De ce fait, vous avez déjà pratiquement toutes les clés pour créer un programme Python exécutable. Le même mécanisme est utilisé pour la création de modules. Les modules sont eux aussi des fichiers contenant du code Python.

Enfin, nous verrons à la fin de ce chapitre comment créer des **packages** pour regrouper nos modules ayant un rapport entre eux.

C'est parti !



Mettre en boîte notre code

Fini, l'interpréteur ?

Je le répète encore, l'interpréteur est véritablement très pratique pour un grand nombre de raisons. Et la meilleure d'entre elles est qu'il propose une manière interactive d'écrire un programme, qui permet de tester le résultat de chaque instruction. Toutefois, l'interpréteur a aussi un défaut : le code que vous saisissez est effacé à la fermeture de la fenêtre. Or, nous commençons à être capables de rédiger des programmes relativement complexes, même si vous ne vous en rendez pas encore compte. Dans ces conditions, devoir réécrire le code entier de son programme à chaque fois qu'on ouvre l'interpréteur de commandes est assez lourd.

La solution ? Mettre notre code dans un fichier que nous pourrons lancer à volonté, comme un véritable programme !

Comme je l'ai dit au début de ce chapitre, il est grand temps que je vous présente cette possibilité. Mais on ne dit pas adieu à l'interpréteur de commandes pour autant. On lui dit juste au revoir pour cette fois... on le retrouvera bien assez tôt, la possibilité de tester le code à la volée est vraiment un atout pour apprendre le langage.

Emprisonnons notre programme dans un fichier

Pour cette démonstration, je reprendrai le code optimisé du programme calculant si une année est bissextile. C'est un petit programme dont l'utilité est certes discutable mais il remplit un but précis, en l'occurrence dire si l'année saisie par l'utilisateur est bissextile ou non : cela suffit pour un premier essai.

Je vous remets le code ici pour que nous travaillions tous sur les mêmes lignes, même si votre version fonctionnera également sans problème dans un fichier, si elle tournait sous l'interpréteur de commandes.

```
# Programme testant si une année, saisie par l'utilisateur, est bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'utilisateur fournisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas saisi un nombre

if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")
```

▷ Copier ce code
Code web : 886842

C'est à votre tour de travailler maintenant, je vais vous donner des pistes mais je ne vais pas me mettre à votre place, chacun prend ses habitudes en fonction de ses préférences.

Ouvrez un éditeur basique : sous Windows, le bloc-notes est candidat, Wordpad ou Word sont exclus ; sous Linux, vous pouvez utiliser Vim ou Emacs. Insérez le code dans ce fichier et enregistrez-le avec l'extension `.py` (exemple `bissextile.py`), comme à la figure 7.1. Cela permettra au système d'exploitation de savoir qu'il doit utiliser Python pour exécuter ce programme[1].

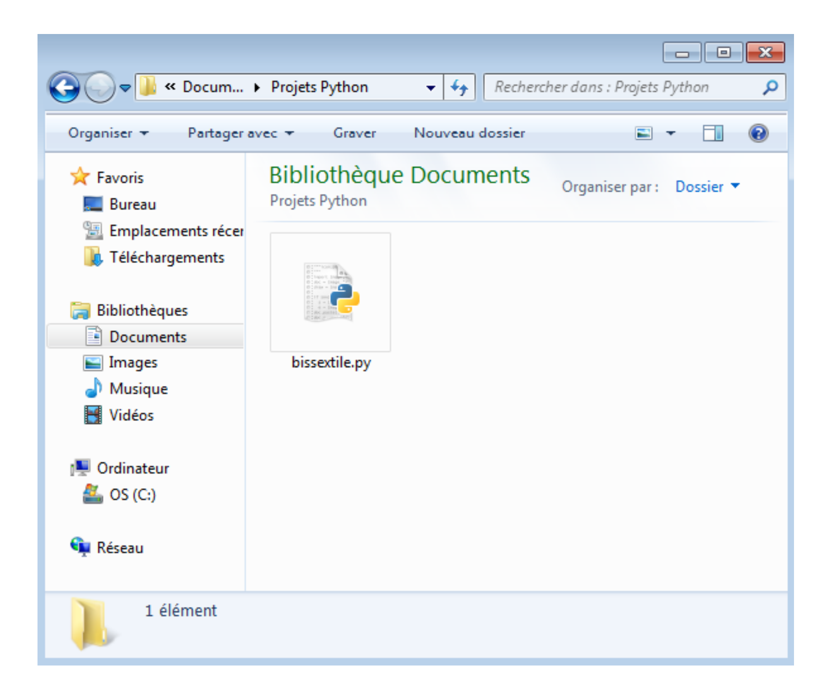


FIGURE 7.1 – Enregistrer un fichier Python sous Windows

Sous Linux, vous devrez ajouter dans votre fichier une ligne, tout au début, spécifiant le chemin de l'interpréteur Python (si vous avez déjà rédigé des scripts, en bash par exemple, cette méthode ne vous surprendra pas). La première ligne de votre programme sera :

```
#!chemin
```

Remplacez alors le terme `chemin` par le chemin donnant accès à l'interpréteur, par exemple : `/usr/bin/python3.2`. Vous devrez changer le droit d'exécution du fichier avant de l'exécuter comme un script.

Sous Windows, rendez-vous dans le dossier où vous avez enregistré votre fichier `.py`. Vous pouvez faire un double-clic dessus, Windows saura qu'il doit appeler Python grâce à l'extension `.py` et Python reprend la main. Attendez toutefois car il reste quelques petites choses à régler avant de pouvoir exécuter votre programme.

1. Cela est nécessaire sous Windows uniquement.

Quelques ajustements

Quand on exécute un programme directement dans un fichier et que le programme contient des accents (et c'est le cas ici), il est nécessaire de préciser à Python l'encodage de ces accents. Je ne vais pas rentrer dans les détails, je vais simplement vous donner une ligne de code qu'il faudra placer tout en haut de votre programme (sous Linux, cette ligne doit figurer juste en-dessous du chemin de l'interpréteur Python).

```
# -*-coding:ENCODAGE -*
```

Sous Windows, vous devrez probablement remplacer `ENCODAGE` par « Latin-1 ». Sous Linux, ce sera plus vraissemblablement « utf-8 ». Ce n'est pas le lieu, ni le moment, pour un cours sur les encodages. Utilisez simplement la ligne qui marche chez vous et tout ira bien.

Il est probable, si vous exécutez votre application d'un double-clic, que votre programme se referme immédiatement après vous avoir demandé l'année. En réalité, il fait bel et bien le calcul mais il arrive à la fin du programme en une fraction de seconde et referme l'application, puisqu'elle est finie. Pour pallier cette difficulté, il faut demander à votre programme de se mettre en pause à la fin de son exécution. Vous devrez rajouter une instruction un peu spéciale, un appel système qui marche sous Windows (pas sous Linux). Il faut tout d'abord importer le module `os`. Ensuite, on rajoute l'appel à la fonction `os.system` en lui passant en paramètre la chaîne de caractères « pause » (cela, à la fin de votre programme). Sous Linux, vous pouvez simplement exécuter votre programme dans la console ou, si vous tenez à faire une pause, utilisez par exemple `input` avant la fin de votre programme (pas bien élégant toutefois).

```
# -*-coding:Latin-1 -*

import os # On importe le module os qui dispose de variables
          # et de fonctions utiles pour dialoguer avec votre
          # système d'exploitation

# Programme testant si une année, saisie par l'utilisateur, est
    bissextile ou non

annee = input("Saisissez une année : ") # On attend que l'
    utilisateur fournisse l'année qu'il désire tester
annee = int(annee) # Risque d'erreur si l'utilisateur n'a pas
    saisi un nombre

if annee % 400 == 0 or (annee % 4 == 0 and annee % 100 != 0):
    print("L'année saisie est bissextile.")
else:
    print("L'année saisie n'est pas bissextile.")

# On met le programme en pause pour éviter qu'il ne se referme
    (Windows)
os.system("pause")
```

Vous pouvez désormais ouvrir votre fichier `bissextile.py`, le programme devrait fonctionner parfaitement (figure 7.2).

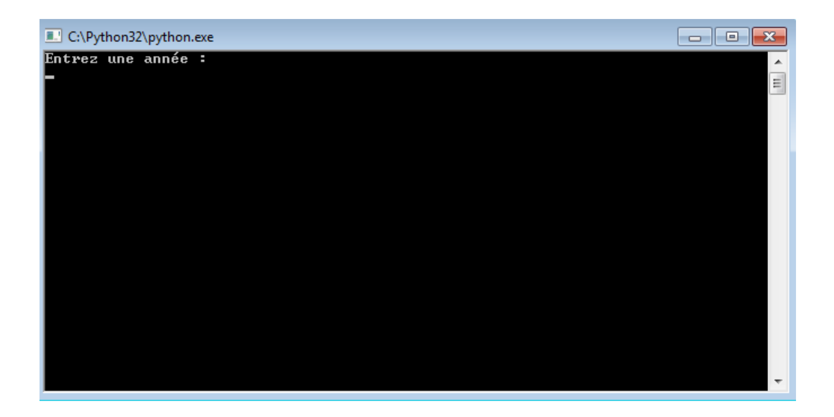


FIGURE 7.2 – Notre programme ne se ferme plus !



Quand vous exécutez ce script, que ce soit sous Windows ou Linux, vous faites toujours appel à l'interpréteur Python ! Votre programme n'est pas compilé mais chaque ligne d'instruction est exécutée à la volée par l'interpréteur, le même qui exécutait vos premiers programmes dans l'interpréteur de commandes. La grande différence ici est que Python exécute votre programme depuis le fichier et que donc, si vous souhaitez modifier le programme, il faudra modifier le fichier.

Sachez qu'il existe des éditeurs spécialisés pour Python, notamment Idle qui est installé en même temps que Python (personnellement je ne l'utilise pas). Vous pouvez l'ouvrir avec un clic droit sur votre fichier `.py` et regarder comment il fonctionne, ce n'est pas bien compliqué et vous pouvez même exécuter votre programme depuis ce logiciel. Mais, étant donné que je ne l'utilise pas, je ne vous ferai pas un cours dessus. Si vous avez du mal à utiliser une des fonctionnalités du logiciel, recherchez sur Internet : d'autres tutoriels doivent exister, en anglais dans le pire des cas.

Je viens pour conquérir le monde... et créer mes propres modules

Mes modules à moi

Bon, nous avons vu le plus dur... ça va ? Rassurez-vous, nous n'allons rien faire de compliqué dans cette dernière section. Le plus dur est derrière nous.

Commencez par vous créer un espace de test pour les petits programmes Python que nous allons être amenés à créer, un joli dossier à l'écart de vos photos et musiques.

Nous allons créer deux fichiers `.py` dans ce dossier :

- un fichier `multipli.py`, qui contiendra la fonction `table` que nous avons codée au chapitre précédent ;
- un fichier `test.py`, qui contiendra le test d'exécution de notre module.

Vous devriez vous en tirer sans problème. N'oubliez pas de spécifier la ligne précisant l'encodage en tête de vos deux fichiers. Maintenant, voyons le code du fichier `multipli.py`.

```
"""module multipli contenant la fonction table"""

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

On se contente de définir une seule fonction, `table`, qui affiche la table de multiplication choisie. Rien de nouveau jusqu'ici. Si vous vous souvenez des `docstrings`, dont nous avons parlé au chapitre précédent, vous voyez que nous en avons inséré une nouvelle ici, non pas pour commenter une fonction mais bien un module entier. C'est une bonne habitude à prendre quand nos projets deviennent importants.

Voici le code du fichier `test.py`, n'oubliez pas la ligne précisant votre encodage, en tête du fichier.

```
import os
from multipli import *

# test de la fonction table
table(3, 20)
os.system("pause")
```

En le lançant directement, voilà ce qu'on obtient :

```
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
11 * 3 = 33
12 * 3 = 36
13 * 3 = 39
14 * 3 = 42
```

```
15 * 3 = 45
16 * 3 = 48
17 * 3 = 51
18 * 3 = 54
19 * 3 = 57
20 * 3 = 60
Appuyez sur une touche pour continuer...
```

Je ne pense pas avoir grand chose à ajouter. Nous avons vu comment créer un module, il suffit de le mettre dans un fichier. On peut alors l'importer depuis un autre fichier *contenu dans le même répertoire* en précisant le nom du fichier (sans l'extension `.py`). Notre code, encore une fois, n'est pas très utile mais vous pouvez le modifier pour le rendre plus intéressant, vous en avez parfaitement les compétences à présent.

Au moment d'importer votre module, Python va lire (ou créer si il n'existe pas) un fichier `.pyc`. À partir de la version 3.2, ce fichier se trouve dans un dossier `__pycache__`.

Ce fichier est généré par Python et contient le code compilé (ou presque) de votre module. Il ne s'agit pas réellement de langage machine mais d'un format que Python décode un peu plus vite que le code que vous pouvez écrire. Python se charge lui-même de générer ce fichier et vous n'avez pas vraiment besoin de vous en soucier quand vous codez, simplement ne soyez pas surpris.

Faire un test de module dans le module-même

Dans l'exemple que nous venons de voir, nous avons créé deux fichiers, le premier contenant un module, le second testant ledit module. Mais on peut très facilement tester le code d'un module dans le module même. Cela veut dire que vous pourriez exécuter votre module comme un programme à lui tout seul, un programme qui testerait le module écrit dans le même fichier. Voyons voir cela.

Reprenons le code du module `multipli` :

```python
"""module multipli contenant la fonction table"""

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1
```

Ce module définit une seule fonction, `table`, qu'il pourrait être bon de tester. Oui mais… si nous rajoutons juste en dessous une ligne, par exemple `table(8)`, cette ligne sera exécutée lors de l'importation et donc, dans le programme appelant le module. Quand vous ferez `import multipli`, vous verrez la table de multiplication par 8 s'afficher… hum, il y a mieux.

Heureusement, il y a un moyen très rapide de séparer les éléments du code qui doivent

être exécutés lorsqu'on lance le module directement en tant que programme ou lorsqu'on cherche à l'importer. Voici le code de la solution, les explications suivent :

```
"""module multipli contenant la fonction table"""

import os

def table(nb, max=10):
    """Fonction affichant la table de multiplication par nb de
    1 * nb jusqu'à max * nb"""
    i = 0
    while i < max:
        print(i + 1, "*", nb, "=", (i + 1) * nb)
        i += 1

# test de la fonction table
if __name__ == "__main__":
    table(4)
    os.system("pause")
```



N'oubliez pas la ligne indiquant l'encodage !

Voilà. À présent, si vous faites un double-clic directement sur le fichier `multipli.py`, vous allez voir la table de multiplication par 4. En revanche, si vous l'importez, le code de test ne s'exécutera pas. Tout repose en fait sur la variable `__name__`, c'est une variable qui existe dès le lancement de l'interpréteur. Si elle vaut `__main__`, cela veut dire que le fichier appelé est le fichier exécuté. Autrement dit, si `__name__` vaut `__main__`, vous pouvez mettre un code qui sera exécuté si le fichier est lancé directement comme un exécutable.

Prenez le temps de comprendre ce mécanisme, faites des tests si nécessaire, cela pourra vous être utile par la suite.

Les packages

Les modules sont un des moyens de regrouper plusieurs fonctions (et, comme on le verra plus tard, certaines classes également). On peut aller encore au-delà en regroupant des modules dans ce qu'on va appeler des **packages**.

En théorie

Comme je l'ai dit, un package sert à regrouper plusieurs modules. Cela permet de ranger plus proprement vos modules, classes et fonctions dans des emplacements séparés. Si vous voulez y accéder, vous allez devoir fournir un chemin vers le module que vous

visez. De ce fait, les risques de conflits de noms sont moins importants et surtout, tout est bien plus ordonné.

Par exemple, imaginons que vous installiez un jour une bibliothèque tierce pour écrire une interface graphique. En s'installant, la bibliothèque ne va pas créer ses dizaines (voire ses centaines) de modules au même endroit. Ce serait un peu désordonné... surtout quand on pense qu'on peut ranger tout cela d'une façon plus claire : d'un côté, on peut avoir les différents objets graphiques de la fenêtre, de l'autres les différents évènements (clavier, souris,...), ailleurs encore les effets graphiques...

Dans ce cas, on va sûrement se retrouver face à un package portant le nom de la bibliothèque. Dans ce package se trouveront probablement d'autres packages, un nommé `evenements`, un autre `objets`, un autre encore `effets`. Dans chacun de ces packages, on pourra trouver soit d'autres packages, soit des modules et dans chacun de ces modules, des fonctions.

Ouf ! Cela nous fait une hiérarchie assez complexe non ? D'un autre côté, c'est tout l'intérêt. Concrètement, pour utiliser cette bibliothèque, on n'est pas obligé de connaître tous ses packages, modules et fonctions (heureusement d'ailleurs !) mais juste ceux dont on a réellement besoin.

En pratique

En pratique, les packages sont... des répertoires ! Dedans peuvent se trouver d'autres répertoires (d'autres packages) ou des fichiers (des modules).

Exemple de hiérarchie

Pour notre bibliothèque imaginaire, la hiérarchie des répertoires et fichiers ressemblerait à cela :

- Un répertoire du nom de la bibliothèque contenant :
  - un répertoire `evenements` contenant :
    - un module `clavier` ;
    - un module `souris` ;
    - ...
  - un répertoire `effets` contenant différents effets graphiques ;
  - un répertoire `objets` contenant les différents objets graphiques de notre fenêtre (boutons, zones de texte, barres de menus...).

Importer des packages

Si vous voulez utiliser, dans votre programme, la bibliothèque fictive que nous venons de voir, vous avez plusieurs moyens qui tournent tous autour des mots-clés `from` et `import` :

```
import nom_bibliotheque
```

Cette ligne importe le package contenant la bibliothèque. Pour accéder aux sous-packages, vous utiliserez un point « . » afin de modéliser le chemin menant au module ou à la fonction que vous voulez utiliser :

```
nom_bibliotheque.evenements # Pointe vers le sous-package evenements
nom_bibliotheque.evenements.clavier # Pointe vers le module clavier
```

Si vous ne voulez importer qu'un seul module (ou qu'une seule fonction) d'un package, vous utiliserez une syntaxe similaire, assez intuitive :

```
from nom_bibliotheque.objets import bouton
```

En fonction des besoins, vous pouvez décider d'importer tout un package, un sous-package, un sous-sous-package... ou bien juste un module ou même une seule fonction. Cela dépendra de vos besoins.

Créer ses propres packages

Si vous voulez créer vos propres packages, commencez par créer, dans le même dossier que votre programme Python, un répertoire portant le nom du package.

Dedans, vous devrez créer un fichier `__init__.py` pour que Python reconnaisse ce répertoire comme un package. Ce fichier peut être vide, je vous montrerai brièvement, un peu plus loin, ce qui doit figurer dedans.

Dans ce répertoire, vous pouvez soit :

- mettre vos modules, vos fichiers à l'extension `.py` ;
- créer des sous-packages de la même façon, en créant un répertoire dans votre package et un fichier `__init__.py`.



Ne mettez pas d'espaces dans vos noms de packages et évitez aussi les caractères spéciaux. Quand vous les utilisez dans vos programmes, ces noms sont traités comme des noms de variables et ils doivent donc obéir aux mêmes règles de nommage.

Dans votre fichier `__init__.py`, vous pouvez écrire du code qui est exécuté quand vous importez votre package. Je ne vais pas vous donner d'exemple concret ici, vous avez déjà bien des choses à retenir.

Un dernier exemple

Voici un dernier exemple, que vous pouvez cette fois faire en même temps que moi pour vous assurer que cela fonctionne.

Dans votre répertoire de code, là où vous mettez vos exemples Python, créez un fichier `.py` que vous appelerez `test_package.py`.

Créez dans le même répertoire un dossier `package`. Dedans, créez un fichier `__init__.py` que vous laisserez vide et un fichier `fonctions.py` dans lequel vous recopierez votre fonction `table`.

Dans votre fichier `test_package.py`, si vous voulez importer votre fonction `table`, vous avez plusieurs solutions :

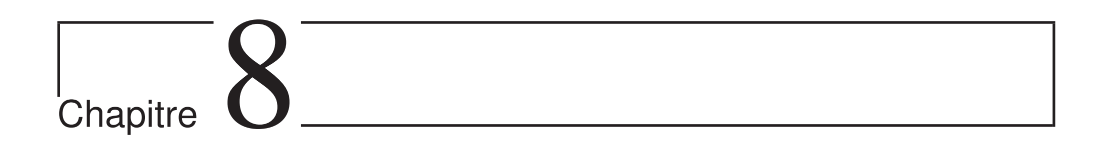
```
from package.fonctions import table
table(5) # Appel de la fonction table

# Ou ...
import package.fonctions
fonctions.table(5) # Appel de la fonction table
```

Voilà. Il reste bien des choses à dire sur les packages mais je crois que vous avez vu l'essentiel. Cette petite explication révélera son importance quand vous aurez à construire des programmes assez volumineux. Évitez de tout mettre dans un seul module sans chercher à hiérarchiser, profitez de cette possibilité offerte par Python.

En résumé

- On peut écrire les programmes Python dans des fichiers portant l'extension `.py`.
- On peut créer des fichiers contenant des **modules** pour séparer le code.
- On peut créer des répertoires contenant des **packages** pour hiérarchiser un programme.



Chapitre 8

Les exceptions

Difficulté : 

Dans ce chapitre, nous aborderons le dernier concept que je considère comme indispensable avant d'attaquer la partie sur la Programmation Orientée Objet, j'ai nommé « les exceptions ».

Comme vous allez le voir, il s'agit des erreurs que peut rencontrer Python en exécutant votre programme. Ces erreurs peuvent être interceptées très facilement et c'est même, dans certains cas, indispensable.

Cependant, il ne faut pas tout intercepter non plus : si Python envoie une erreur, c'est qu'il y a une raison. Si vous ignorez une erreur, vous risquez d'avoir des résultats très étranges dans votre programme.



À quoi cela sert-il ?

Nous avons déjà été confrontés à des erreurs dans nos programmes, certaines que j'ai volontairement provoquées, mais la plupart que vous avez dû rencontrer si vous avez testé un minimum des instructions dans l'interpréteur. Quand Python rencontre une erreur dans votre code, il **lève une exception**. Sans le savoir, vous avez donc déjà vu des exceptions levées par Python :

```
>>> # Exemple classique : test d'une division par zéro
>>> variable = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Attardons-nous sur la dernière ligne. Nous y trouvons deux informations :

- `ZeroDivisionError` : le type de l'exception ;
- `int division or modulo by zero` : le message qu'envoie Python pour vous aider à comprendre l'erreur qui vient de se produire.

Python lève donc des exceptions quand il trouve une erreur, soit dans le code (une erreur de syntaxe, par exemple), soit dans l'opération que vous lui demandez de faire.

Notez qu'à l'instar des variables, on trouve différents types d'exceptions que Python va utiliser en fonction de la situation. Le type d'exception `ValueError`, notamment, pourra être levé par Python face à diverses erreurs de « valeurs ». Dans ce cas, c'est donc le message qui vous indique plus clairement le problème. Nous verrons dans la prochaine partie, consacrée à la Programmation Orientée Objet, ce que sont réellement ces types d'exceptions.

Bon, c'est bien joli d'avoir cette exception. On voit le fichier et la ligne à laquelle s'est produite l'erreur (très pratique quand on commence à travailler sur un projet) et on a une indication sur le problème qui suffit en général à le régler. Mais Python permet quelque chose de bien plus pratique.

Admettons que certaines erreurs puissent être provoquées par l'utilisateur. Par exemple, on demande à l'utilisateur de saisir au clavier un entier et il tape une chaîne de caractères... problème. Nous avons déjà rencontré cette situation : souvenez-vous du programme `bissextile`.

```
annee = input() # On demande à l'utilisateur de saisir l'année
annee = int(annee) # On essaye de convertir l'année en un
    entier
```

Je vous avais dit que si l'utilisateur fournissait ici une valeur impossible à convertir en entier (une lettre par exemple), le programme plantait. En fait, il lève une exception et Python arrête l'exécution du programme. Si vous testez le programme en faisant un double-clic directement dans l'explorateur, il va se fermer tout de suite (en fait, il affiche bel et bien l'erreur mais se referme aussitôt).

Dans ce cas, et dans d'autres cas similaires, Python permet de tester un extrait de code. S'il ne renvoie aucune erreur, Python continue. Sinon, on peut lui demander d'exécuter une autre action (par exemple, redemander à l'utilisateur de saisir l'année). C'est ce que nous allons voir ici.

Forme minimale du bloc `try`

On va parler ici de bloc `try`. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc. Sans plus attendre, voici la syntaxe :

```
try:
    # Bloc à essayer
except:
    # Bloc qui sera exécuté en cas d'erreur
```

Dans l'ordre, nous trouvons :

- Le mot-clé `try` suivi des deux points « : » (*try* signifie « essayer » en anglais).
- Le bloc d'instructions à essayer.
- Le mot-clé `except` suivi, une fois encore, des deux points « : ». Il se trouve au même niveau d'indentation que le `try`.
- Le bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc.

Reprenons notre test de conversion en enfermant dans un bloc `try` l'instruction susceptible de lever une exception.

```
annee = input()
try: # On essaye de convertir l'année en entier
    annee = int(annee)
except:
    print("Erreur lors de la conversion de l'année.")
```

Vous pouvez tester ce code en précisant plusieurs valeurs différentes pour la variable `annee`, comme « 2010 » ou « annee2010 ».

Dans le titre de cette section, j'ai parlé de *forme minimale* et ce n'est pas pour rien. D'abord, il va de soi que vous ne pouvez intégrer cette solution directement dans votre code. En effet, si l'utilisateur saisit une année impossible à convertir, le système affiche certes une erreur mais finit par planter (puisque l'année, au final, n'a pas été convertie). Une des solutions envisageables est d'attribuer une valeur par défaut à l'année, en cas d'erreur, ou de redemander à l'utilisateur de saisir l'année.

Ensuite et surtout, cette méthode est assez grossière. Elle essaye une instruction et intercepte *n'importe quelle* exception liée à cette instruction. Ici, c'est acceptable car nous n'avons pas énormément d'erreurs possibles sur cette instruction. Mais c'est une mauvaise habitude à prendre. Voici une manière plus élégante et moins dangereuse.

Forme plus complète

Nous allons apprendre à compléter notre bloc `try`. Comme je l'ai indiqué plus haut, la forme minimale est à éviter pour plusieurs raisons.

D'abord, elle ne différencie pas les exceptions qui pourront être levées dans le bloc `try`. Ensuite, Python peut lever des exceptions qui ne signifient pas nécessairement qu'il y a eu une erreur.

Exécuter le bloc except pour un type d'exception précis

Dans l'exemple que nous avons vu plus haut, on ne pense qu'à un type d'exceptions susceptible d'être levé : le type `ValueError`, qui trahirait une erreur de conversion. Voyons un autre exemple :

```
try:
    resultat = numerateur / denominateur
except:
    print("Une erreur est survenue... laquelle ?")
```

Ici, plusieurs erreurs sont susceptibles d'intervenir, chacune levant une exception différente.

- `NameError` : l'une des variables `numerateur` ou `denominateur` n'a pas été définie (elle n'existe pas). Si vous essayez dans l'interpréteur l'instruction `print(numerateur)` alors que vous n'avez pas défini la variable `numerateur`, vous aurez la même erreur.
- `TypeError` : l'une des variables `numerateur` ou `denominateur` ne peut diviser ou être divisée (les chaînes de caractères ne peuvent être divisées, ni diviser d'autres types, par exemple). Cette exception est levée car vous utilisez l'opérateur de division « / » sur des types qui ne savent pas quoi en faire.
- `ZeroDivisionError` : encore elle ! Si `denominateur` vaut 0, cette exception sera levée.

Cette énumération n'est pas une liste exhaustive de toutes les exceptions qui peuvent être levées à l'exécution de ce code. Elle est surtout là pour vous montrer que plusieurs erreurs peuvent se produire sur une instruction (c'est encore plus flagrant sur un bloc constitué de plusieurs instructions) et que la forme minimale intercepte toutes ces erreurs sans les distinguer, ce qui peut être problématique dans certains cas.

Tout se joue sur la ligne du `except`. Entre ce mot-clé et les deux points, vous pouvez préciser le type de l'exception que vous souhaitez traiter.

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été d
        éfinie.")
```

Ce code ne traite que le cas où une exception `NameError` est levée. On peut intercepter les autres types d'exceptions en créant d'autres blocs `except` à la suite :

```python
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
except TypeError:
    print("La variable numerateur ou denominateur possède un type incompatible avec la division.")
except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")
```

C'est mieux non ?

Allez un petit dernier !

On peut capturer l'exception et afficher son message grâce au mot-clé `as` que vous avez déjà vu dans un autre contexte (si si, rappelez-vous de l'importation de modules).

```python
try:
    # Bloc de test
except type_de_l_exception as exception_retournee:
    print("Voici l'erreur :", exception_retournee)
```

Dans ce cas, une variable `exception_retournee` est créée par Python si une exception du type précisé est levée dans le bloc `try`.

Je vous conseille de *toujours* préciser un type d'exceptions après `except` (sans nécessairement capturer l'exception dans une variable, bien entendu). D'abord, vous ne devez pas utiliser `try` comme une méthode miracle pour tester n'importe quel bout de code. Il est important que vous gardiez le maximum de contrôle sur votre code. Cela signifie que, si une erreur se produit, vous devez être capable de l'anticiper. En pratique, vous n'irez pas jusqu'à tester si une variable quelconque existe bel et bien, il faut faire un minimum confiance à son code. Mais si vous êtes en face d'une division et que le dénominateur pourrait avoir une valeur de 0, placez la division dans un bloc `try` et précisez, après le `except`, le type de l'exception qui risque de se produire (`ZeroDivisionError` dans cet exemple).

Si vous adoptez la forme minimale (à savoir `except` sans préciser un type d'exception qui pourrait se produire sur le bloc `try`), toutes les exceptions seront traitées de la même façon. Et même si `exception = erreur` la plupart du temps, ce n'est pas toujours le cas. Par exemple, Python lève une exception quand vous voulez fermer votre programme avec le raccourci CTRL + C. Ici vous ne voyez peut-être pas le problème mais si votre bloc `try` est dans une boucle, vous ne pourrez pas arrêter votre programme avec CTRL + C, puisque l'exception sera traitée par votre `except`.

Je vous conseille donc de toujours préciser un type d'exception possible après votre `except`. Vous pouvez bien entendu faire des tests dans l'interpréteur de commandes Python pour reproduire l'exception que vous voulez traiter et ainsi connaître son type.

Les mots-clés `else` et `finally`

Ce sont deux mots-clés qui vont nous permettre de construire un bloc `try` plus complet.

Le mot-clé `else`

Vous avez déjà vu ce mot-clé et j'espère que vous vous en rappelez. Dans un bloc `try`, `else` va permettre d'exécuter une action si aucune erreur ne survient dans le bloc. Voici un petit exemple :

```
try:
    resultat = numerateur / denominateur
except NameError:
    print("La variable numerateur ou denominateur n'a pas été définie.")
except TypeError:
    print("La variable numerateur ou denominateur possède un type incompatible avec la division.")
except ZeroDivisionError:
    print("La variable denominateur est égale à 0.")
else:
    print("Le résultat obtenu est", resultat)
```

Dans les faits, on utilise assez peu `else`. La plupart des codeurs préfère mettre la ligne contenant le `print` directement dans le bloc `try`. Pour ma part, je trouve que c'est important de distinguer entre le bloc `try` et ce qui s'effectue ensuite. La ligne du `print` ne produira vraisemblablement aucune erreur, inutile de la placer dans le bloc `try`.

Le mot-clé `finally`

`finally` permet d'exécuter du code après un bloc `try`, *quelle que soit le résultat de l'exécution dudit bloc*. La syntaxe est des plus simples :

```
try:
    # Test d'instruction(s)
except TypeDInstruction:
    # Traitement en cas d'erreur
finally:
    # Instruction(s) exécutée(s) qu'il y ait eu des erreurs ou non
```



Est-ce que cela ne revient pas au même si on met du code juste après le bloc ?

Pas tout à fait. Le bloc `finally` est exécuté dans tous les cas de figures. Quand bien même Python trouverait une instruction `return` dans votre bloc `except` par exemple, il exécutera le bloc `finally`.

Un petit bonus : le mot-clé `pass`

Il peut arriver, dans certains cas, que l'on souhaite tester un bloc d'instructions. . . mais ne rien faire en cas d'erreur. Toutefois, un bloc `try` ne peut être seul.

```
>>> try:
...     1/0
...
  File "<stdin>", line 3

    ^
SyntaxError: invalid syntax
```

Il existe un mot-clé que l'on peut utiliser dans ce cas. Son nom est `pass` et sa syntaxe est très simple d'utilisation :

```
try:
    # Test d'instruction(s)
except type_de_l_exception: # Rien ne doit se passer en cas d'
    erreur
    pass
```

Je ne vous encourage pas particulièrement à utiliser ce mot-clé mais il existe, et vous le savez à présent.

`pass` n'est pas un mot-clé propre aux exceptions : on peut également le trouver dans des conditions ou dans des fonctions que l'on souhaite laisser vides.

Voilà, nous avons vu l'essentiel. Il nous reste à faire un petit point sur les assertions et à voir comment lever une exception (ce sera très rapide).

Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs `try ... except`.

Voyons comment cela fonctionne : nous allons pour l'occasion découvrir un nouveau mot-clé (encore un), `assert`. Sa syntaxe est la suivante :

```
assert test
```

Si le test renvoie `True`, l'exécution se poursuit normalement. Sinon, une exception `AssertionError` est levée.

Voyons un exemple :

```
>>> var = 5
>>> assert var == 5
>>> assert var == 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
6 AssertionError
7 >>>
```

Comme vous le voyez, la ligne 2 s'exécute sans problème et ne lève aucune exception. On teste en effet si `var == 5`. C'est le cas, le test est donc vrai, aucune exception n'est levée.

À la ligne suivante, cependant, le test est `var == 8`. Cette fois, le test est faux et une exception du type `AssertionError` est levée.



À quoi cela sert-il, concrètement ?

Dans le programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur ne saisit pas une année inférieure ou égale à 0 par exemple. Avec les assertions, c'est très facile à faire :

```
annee = input("Saisissez une année supérieure à 0 :")
try:
    annee = int(annee) # Conversion de l'année
    assert annee > 0
except ValueError:
    print("Vous n'avez pas saisi un nombre.")
except AssertionError:
    print("L'année saisie est inférieure ou égale à 0.")
```

Lever une exception

Hmmm... je vois d'ici les mines sceptiques (non non, ne vous cachez pas !). Vous vous demandez probablement pourquoi vous feriez le boulot de Python en levant des exceptions. Après tout, votre travail, c'est en théorie d'éviter que votre programme plante.

Parfois, cependant, il pourra être utile de lever des exceptions. Vous verrez tout l'intérêt du concept quand vous créerez vos propres classes... mais ce n'est pas pour tout de suite. En attendant, je vais vous donner la syntaxe et vous pourrez faire quelques tests, vous verrez de toute façon qu'il n'y a rien de compliqué.

On utilise un nouveau mot-clé pour lever une exception... le mot-clé `raise`.

```
raise TypeDeLException("message à afficher")
```

Prenons un petit exemple, toujours autour de notre programme `bissextile`. Nous allons lever une exception de type `ValueError` si l'utilisateur saisit une année négative ou nulle.

```
annee = input() # L'utilisateur saisit l'année
try:
    annee = int(annee) # On tente de convertir l'année
```

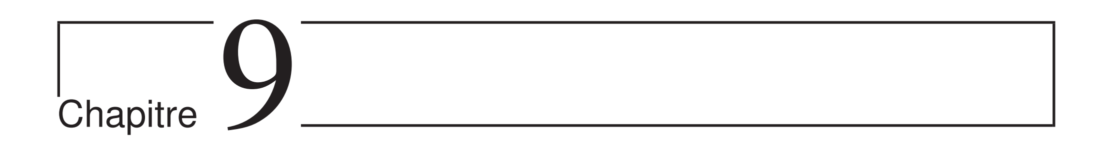
```
    if annee<=0:
        raise ValueError("l'année saisie est négative ou nulle"
            )
except ValueError:
    print("La valeur saisie est invalide (l'année est peut-être
         négative).")
```

Ce que nous venons de faire est réalisable sans l'utilisation des exceptions mais c'était surtout pour vous montrer la syntaxe dans un véritable contexte. Ici, on lève une exception que l'on intercepte immédiatement ou presque, l'intérêt est donc limité. Bien entendu, la plupart du temps ce n'est pas le cas.

Il reste des choses à découvrir sur les exceptions, mais on en a assez fait pour ce chapitre et cette partie. Je ne vous demande pas de connaître toutes les exceptions que Python est amené à utiliser (certaines d'entre elles pourront d'ailleurs n'exister que dans certains modules). En revanche, vous devez être capables de savoir, grâce à l'interpréteur de commandes, quelles exceptions peuvent être levées par Python dans une situation donnée.

En résumé

- On peut intercepter les erreurs (ou exceptions) levées par notre code grâce aux blocs `try except`.
- La syntaxe d'une assertion est `assert test:`.
- Les assertions lèvent une exception `AssertionError` si le test échoue.
- On peut lever une exception grâce au mot-clé `raise` suivi du type de l'exception.


Chapitre 9

TP : tous au ZCasino

Difficulté : 

L'heure de vérité a sonné ! C'est dans ce premier TP que je vais faire montre de ma cruauté sans limite en vous lâchant dans la nature... ou presque. Ce n'est pas tout à fait votre premier TP, dans le sens où le programme du chapitre 4, sur les conditions, constituait votre première expérience en la matière. Mais, à ce moment-là, nous n'avions pas fait un programme très... récréatif.

Cette fois, nous allons nous atteler au développement d'un petit jeu de casino. Vous trouverez le détail de l'énoncé plus bas, ainsi que quelques conseils pour la réalisation de ce TP.

Si, durant ce TP, vous sentez que certaines connaissances vous manquent, revenez en arrière ; prenez tout votre temps, on n'est pas pressé !



Notre sujet

Dans ce chapitre, nous allons essayer de faire un petit programme que nous appellerons ZCasino. Il s'agira d'un petit jeu de roulette très simplifié dans lequel vous pourrez miser une certaine somme et gagner ou perdre de l'argent (telle est la fortune, au casino !). Quand vous n'avez plus d'argent, vous avez perdu.

Notre règle du jeu

Bon, la roulette, c'est très sympathique comme jeu, mais un peu trop compliqué pour un premier TP. Alors, on va simplifier les règles et je vous présente tout de suite ce que l'on obtient :

- Le joueur mise sur un numéro compris entre 0 et 49 (50 numéros en tout). En choisissant son numéro, il y dépose la somme qu'il souhaite miser.
- La roulette est constituée de 50 cases allant naturellement de 0 à 49. Les numéros pairs sont de couleur noire, les numéros impairs sont de couleur rouge. Le croupier lance la roulette, lâche la bille et quand la roulette s'arrête, relève le numéro de la case dans laquelle la bille s'est arrêtée. Dans notre programme, nous ne reprendrons pas tous ces détails « matériels » mais ces explications sont aussi à l'intention de ceux qui ont eu la chance d'éviter les salles de casino jusqu'ici. Le numéro sur lequel s'est arrêtée la bille est, naturellement, le numéro gagnant.
- Si le numéro gagnant est celui sur lequel le joueur a misé (probabilité de 1/50, plutôt faible), le croupier lui remet 3 fois la somme misée.
- Sinon, le croupier regarde si le numéro misé par le joueur est de la même couleur que le numéro gagnant (s'ils sont tous les deux pairs ou tous les deux impairs). Si c'est le cas, le croupier lui remet 50 % de la somme misée. Si ce n'est pas le cas, le joueur perd sa mise.

Dans les deux scénarios gagnants vus ci-dessus (le numéro misé et le numéro gagnant sont identiques ou ont la même couleur), le croupier remet au joueur la somme initialement misée avant d'y ajouter ses gains. Cela veut dire que, dans ces deux scénarios, le joueur récupère de l'argent. Il n'y a que dans le troisième cas qu'il perd la somme misée[1].

Organisons notre projet

Pour ce projet, nous n'allons pas écrire de module. Nous allons utiliser ceux de Python, qui sont bien suffisants pour l'instant, notamment celui permettant générer de l'aléatoire, que je vais présenter plus bas. En attendant, ne vous privez quand même pas de créer un répertoire et d'y mettre le fichier `ZCasino.py`, tout va se jouer ici.

Vous êtes capables d'écrire le programme `ZCasino` tel qu'expliqué dans la première partie sans difficulté… sauf pour générer des nombres aléatoires. Python a dédié tout

1. On utilisera pour devise le dollar $ à la place de l'euro pour des raisons d'encodage sous la console Windows.

un module à la génération d'éléments pseudo-aléatoires, le module `random`.

Le module `random`

Dans ce module, nous allons nous intéresser particulièrement à la fonction `randrange` qui peut s'utiliser de deux manières :

- en ne précisant qu'un paramètre (`randrange(6)` renvoie un nombre aléatoire compris entre 0 et 5) ;
- en précisant deux paramètres (`randrange(1, 7)` : renvoie un nombre aléatoire compris entre 1 et 6, ce qui est utile, par exemple, pour reproduire une expérience avec un dé à six faces).

Pour tirer un nombre aléatoire compris entre 0 et 49 et simuler ainsi l'expérience du jeu de la roulette, nous allons donc utiliser l'instruction `randrange(50)`.

Il existe d'autres façons d'utiliser `randrange` mais nous n'en aurons pas besoin ici et je dirais même que, pour ce programme, seule la première utilisation vous sera utile.

N'hésitez pas à faire des tests dans l'interpréteur de commandes (vous n'avez pas oublié où c'est, hein ?) et essayez plusieurs syntaxes de la fonction `randrange`. Je vous rappelle qu'elle se trouve dans le module `random`, n'oubliez pas de l'importer.

Arrondir un nombre

Vous l'avez peut-être bien noté, dans l'explication des règles je spécifiais que si le joueur misait sur la bonne couleur, il obtenait 50% de sa mise. Oui mais... c'est quand même mieux de travailler avec des entiers. Si le joueur mise 3$, par exemple, on lui rend 1,5$. C'est encore acceptable mais, si cela se poursuit, on risque d'arriver à des nombres flottants avec beaucoup de chiffres après la virgule. Alors autant arrondir au nombre supérieur. Ainsi, si le joueur mise 3$, on lui rend 2$. Pour cela, on va utiliser une fonction du module `math` nommée `ceil`. Je vous laisse regarder ce qu'elle fait, il n'y a rien de compliqué.

À vous de jouer

Voilà, vous avez toutes les clés en main pour coder ce programme. Prenez le temps qu'il faut pour y arriver, ne vous ruez pas sur la correction, le but du TP est que vous appreniez à coder vous-mêmes un programme... et celui-ci n'est pas très difficile. Si vous avez du mal, morcelez le programme, ne codez pas tout d'un coup. Et n'hésitez pas à passer par l'interpréteur pour tester des fonctionnalités : c'est réellement une chance qui vous est donnée, ne la laissez pas passer.

À vous de jouer !

Correction !

C'est encore une fois l'heure de comparer nos versions. Et, une fois encore, il est très peu probable que vous ayez un code identique au mien. Donc si le vôtre fonctionne, je dirais que c'est l'essentiel. Si vous vous heurtez à des difficultés insurmontables, la correction est là pour vous aider.



. . . ATTENTION. . . voici. . . la solution !

```python
# Ce fichier abrite le code du ZCasino, un jeu de roulette adapté

import os
from random import randrange
from math import ceil

# Déclaration des variables de départ
argent = 1000 # On a 1000 $ au début du jeu
continuer_partie = True # Booléen qui est vrai tant qu'on doit
                        # continuer la partie

print("Vous vous installez à la table de roulette avec", argent, "$.")

while continuer_partie: # Tant qu'on doit continuer la partie
    # on demande à l'utilisateur de saisir le nombre sur
    # lequel il va miser
    nombre_mise = -1
    while nombre_mise < 0 or nombre_mise > 49:
        nombre_mise = input("Tapez le nombre sur lequel vous voulez miser (entre 0 et 49) : ")
        # On convertit le nombre misé
        try:
            nombre_mise = int(nombre_mise)
        except ValueError:
            print("Vous n'avez pas saisi de nombre")
            nombre_mise = -1
            continue
        if nombre_mise < 0:
            print("Ce nombre est négatif")
        if nombre_mise > 49:
            print("Ce nombre est supérieur à 49")

    # À présent, on sélectionne la somme à miser sur le nombre
    mise = 0
    while mise <= 0 or mise > argent:
        mise = input("Tapez le montant de votre mise : ")
```

```python
        # On convertit la mise
        try:
            mise = int(mise)
        except ValueError:
            print("Vous n'avez pas saisi de nombre")
            mise = -1
            continue
        if mise <= 0:
            print("La mise saisie est négative ou nulle.")
        if mise > argent:
            print("Vous ne pouvez miser autant, vous n'avez que ", argent, "$")

    # Le nombre misé et la mise ont été sélectionnés par
    # l'utilisateur, on fait tourner la roulette
    numero_gagnant = randrange(50)
    print("La roulette tourne... ... et s'arrête sur le numéro", numero_gagnant)

    # On établit le gain du joueur
    if numero_gagnant == nombre_mise:
        print("Félicitations ! Vous obtenez", mise * 3, "$ !")
        argent += mise * 3
    elif numero_gagnant % 2 == nombre_mise % 2: # ils sont de la même couleur
        mise = ceil(mise * 0.5)
        print("Vous avez misé sur la bonne couleur. Vous obtenez", mise, "$")
        argent += mise
    else:
        print("Désolé l'ami, c'est pas pour cette fois. Vous perdez votre mise.")
        argent -= mise

    # On interrompt la partie si le joueur est ruiné
    if argent <= 0:
        print("Vous êtes ruiné ! C'est la fin de la partie.")
        continuer_partie = False
    else:
        # On affiche l'argent du joueur
        print("Vous avez à présent", argent, "$")
        quitter = input("Souhaitez-vous quitter le casino (o/n) ? ")
        if quitter == "o" or quitter == "O":
            print("Vous quittez le casino avec vos gains.")
            continuer_partie = False

# On met en pause le système (Windows)
os.system("pause")
```

Pour accéder en ligne au code source de la solution, utilisez le code web suivant :

▷ Copier ce code
Code web : 366476 .

Encore une fois, n'oubliez pas la ligne spécifiant l'encodage si vous voulez éviter les surprises.

Une petite chose qui pourrait vous surprendre est la construction des boucles pour tester si le joueur a saisi une valeur correcte (quand on demande à l'utilisateur de taper un nombre entre 0 et 49 par exemple, il faut s'assurer qu'il l'a bien fait). C'est assez simple en vérité : on attend que le joueur saisisse un nombre. Si le nombre n'est pas valide, on demande à nouveau au joueur de saisir ce nombre. J'en ai profité pour utiliser le concept des exceptions afin de vérifier que l'utilisateur saisit bien un nombre. Comme vous l'avez vu, si ce n'est pas le cas, on affiche un message d'erreur. La valeur de la variable qui contient le nombre est remise à `-1` (c'est-à-dire une valeur qui indique à la boucle que nous n'avons toujours pas obtenu de l'utilisateur une valeur valides) et on utilise le mot-clé `continue` pour passer les autres instructions du bloc (sans quoi vous verriez s'afficher un autre message indiquant que le nombre saisi est négatif… c'est plus pratique ainsi). De cette façon, si l'utilisateur fournit une donnée inconvertible, le jeu ne plante pas et lui redemande tout simplement de taper une valeur valide.

La boucle principale fonctionne autour d'un booléen. On utilise une variable nommée `continuer_partie` qui vaut « vrai » tant qu'on doit continuer la partie. Une fois que la partie doit s'interrompre, elle passe à « faux ». Notre boucle globale, qui gère le déroulement de la partie, travaille sur ce booléen ; par conséquent, dès qu'il passe à la valeur « faux », la boucle s'interrompt et le programme se met en pause. Tout le reste, vous devriez le comprendre sans aide, les commentaires sont là pour vous expliquer. Si vous avez des doutes, vous pouvez tester les lignes d'instructions problématiques dans votre interpréteur de commandes Python : encore une fois, n'oubliez pas cet outil.

Et maintenant ?

Prenez bien le temps de lire ma version et surtout de modifier la vôtre, si vous êtes arrivés à une version qui fonctionne bien ou qui fonctionne presque. Ne mettez pas ce projet à la corbeille sous prétexte que nous avons fini de le coder et qu'il marche. On peut toujours améliorer un projet et celui-ci ne fait évidemment pas exception. Vous trouverez probablement de nouveaux concepts, dans la suite de ce livre, qui pourront être utilisés dans le programme de `ZCasino`.

Deuxième partie

La Programmation Orientée Objet côté utilisateur

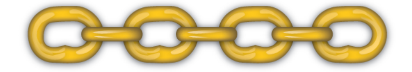
Chapitre 10

Notre premier objet : les chaînes de caractères

Difficulté : 

Les objets... vaste sujet ! Avant d'en créer, nous allons d'abord voir de quoi il s'agit. Nous allons commencer avec les chaînes de caractères, un type que vous pensez bien connaître.

Dans ce chapitre, vous allez découvrir petit à petit le mécanisme qui se cache derrière la notion d'objet. Ces derniers font partie des notions incontournables en Python, étant donné que tout ce que nous avons utilisé jusqu'ici... est un objet !



Vous avez dit objet ?

La première question qui risque de vous empêcher de dormir si je n'y réponds pas tout de suite, c'est :



Mais c'est quoi un objet ?

Eh bien, j'ai lu beaucoup de définitions très différentes et je n'ai pas trouvé de point commun à toutes ces définitions. Nous allons donc partir d'une définition incomplète mais qui suffira pour l'instant : **un objet est une structure de données, comme les variables, qui peut contenir elle-même d'autres variables et fonctions.** On étoffera plus loin cette définition, elle suffit bien pour le moment.



Je ne comprends rien. Passe encore qu'une variable en contienne d'autres, après tout les chaînes de caractères contiennent bien des caractères, mais qu'une variable contienne des fonctions... Cela rime à quoi ?

Je pourrais passer des heures à expliquer la théorie du concept que vous n'en seriez pas beaucoup plus avancés. J'ai choisi de vous montrer les objets par l'exemple et donc, vous allez très rapidement voir ce que tout cela signifie. Mais vous allez devoir me faire confiance, au début, sur l'utilité de la méthode objet.

Avant d'attaquer, une petite précision. J'ai dit qu'un objet était un peu comme une variable... en fait, pour être exact, il faut dire qu'une variable est un objet. Toutes les variables avec lesquelles nous avons travaillé jusqu'ici sont des objets. Les fonctions que nous avons vues sont également des objets. *En Python, tout est objet* : gardez cela à l'esprit.

Les méthodes de la classe `str`

Oh la la, j'en vois qui grimacent rien qu'en lisant le titre. Vous n'avez pourtant aucune raison de vous inquiéter ! On va y aller tout doucement.

Posons un problème : comment peut-on passer une chaîne de caractères en minuscules ? Si vous n'avez lu jusqu'à présent que les premiers chapitres, vous ne pourrez pas faire cet exercice ; j'ai volontairement évité de trop aborder les chaînes de caractères jusqu'ici. Mais admettons que vous arriviez à coder une fonction prenant en paramètre la chaîne en question. Vous aurez un code qui ressemble à ceci :

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> mettre_en_minuscule(chaine)
'ne crie pas si fort !'
```

Sachez que, dans les anciennes versions de Python, il y avait un module spécialisé dans

le traitement des chaînes de caractères. On importait ce module et on pouvait appeler la fonction passant une chaîne en minuscules. Ce module existe d'ailleurs encore et reste utilisé pour certains traitements spécifiques. Mais on va découvrir ici une autre façon de faire. Regardez attentivement :

```
>>> chaine = "NE CRIE PAS SI FORT !"
>>> chaine.lower() # Mettre la chaîne en minuscule
'ne crie pas si fort !'
```

La fonction lower est une nouveauté pour vous. Vous devez reconnaître le point « . » qui symbolisait déjà, dans le chapitre sur les modules, une relation d'appartenance (a.b signifiait que b était contenu dans a). Ici, il possède la même signification : la fonction lower est une fonction de la variable chaine.

La fonction lower est propre aux chaînes de caractères. Toutes les chaînes peuvent y faire appel. Si vous tapez type(chaine) dans l'interpréteur, vous obtenez <class 'str'>. Nous avons dit qu'une variable est issue d'un type de donnée. Je vais à présent reformuler : un **objet** est issu d'une **classe**. La **classe** est une forme de type de donnée, sauf qu'elle permet de définir des fonctions et variables propres au type. C'est pour cela que, dans toutes les chaînes de caractères, on peut appeler la fonction lower. C'est tout simplement parce que la fonction lower a été définie dans la classe str. Les fonctions définies dans une classe sont appelées des **méthodes**.

Récapitulons. Nous avons découvert :

– Les **objets**, que j'ai présentés comme des variables, pouvant contenir d'autres variables ou fonctions (que l'on appelle **méthodes**). On appelle une méthode d'un objet grâce à objet.methode().
– Les **classes**, que j'ai présentées comme des types de données. Une classe est un modèle qui servira à construire un objet ; c'est dans la classe qu'on va définir les méthodes propres à l'objet.

Voici le mécanisme qui vous permet d'appeler la méthode lower d'une chaîne :

1. Les développeurs de Python ont créé la classe str qui est utilisée pour créer des chaînes de caractères. Dans cette classe, ils ont défini plusieurs méthodes, comme lower, qui pourront être utilisées par n'importe quel objet construit sur cette classe.

2. Quand vous écrivez chaine = "NE CRIE PAS SI FORT !", Python reconnaît qu'il doit créer une chaîne de caractères. Il va donc créer un objet d'après la classe (le modèle) qui a été définie à l'étape précédente.

3. Vous pouvez ensuite appeler toutes les méthodes de la classe str depuis l'objet chaine que vous venez de créer.

Ouf ! Cela fait beaucoup de choses nouvelles, du vocabulaire et des concepts un peu particuliers.

Vous ne voyez peut-être pas encore tout l'intérêt d'avoir des méthodes définies dans une certaine classe. Cela permet d'abord de bien séparer les diverses fonctionnalités

(on ne peut pas passer en minuscules un nombre entier, cela n'a aucun sens). Ensuite, c'est plus intuitif, une fois passé le choc de la première rencontre.

Bon, on parle, on parle, mais on ne code pas beaucoup !

Mettre en forme une chaîne

Non, vous n'allez pas apprendre à mettre une chaîne en gras, souligné, avec une police Verdana de 15px... Nous ne sommes encore que dans une console. Nous venons de présenter `lower`, il existe d'autres méthodes. Avant tout, voyons un contexte d'utilisation.

Certains d'entre vous se demandent peut-être l'intérêt de passer des chaînes en minuscules... alors voici un petit exemple.

```
chaine = str() # Crée une chaîne vide
# On aurait obtenu le même résultat en tapant chaine = ""

while chaine.lower() != "q":
    print("Tapez 'Q' pour quitter...")
    chaine = input()

print("Merci !")
```

Vous devez comprendre rapidement ce programme. Dans une boucle, on demande à l'utilisateur de taper la lettre « q » pour quitter. Tant que l'utilisateur saisit une autre lettre, la boucle continue de s'exécuter. Dès que l'utilisateur appuie sur la touche Q de son clavier, la boucle s'arrête et le programme affiche « Merci ! ». Cela devrait vous rappeler quelque chose... direction le TP de la partie 1 pour ceux qui ont la mémoire courte.

La petite nouveauté réside dans le test de la boucle : `chaine.lower() != "q"`. On prend la chaîne saisie par l'utilisateur, on la passe en minuscules et on regarde si elle est différente de « q ». Cela veut dire que l'utilisateur peut taper « q » en majuscule ou en minuscule, dans les deux cas la boucle s'arrêtera.

Notez que `chaine.lower()` renvoie la chaîne en minuscules mais ne modifie pas la chaîne. C'est très important, nous verrons pourquoi dans le prochain chapitre.

Notez aussi que nous avons appelé la fonction `str` pour créer une chaîne vide. Je ne vais pas trop compliquer les choses mais sachez qu'appeler ainsi un type en tant que fonction permet de créer un objet de la classe. Ici, `str()` crée un objet *chaîne de caractères*. Nous avons vu dans la première partie le mot-clé `int()`, qui crée aussi un entier[1].

Bon, voyons d'autres méthodes. Je vous invite à tester mes exemples (ils sont commentés, mais on retient mieux en essayant par soi-même).

```
>>> minuscules = "une chaine en minuscules"
>>> minuscules.upper() # Mettre en majuscules
'UNE CHAINE EN MINUSCULES'
```

1. Si nécessaire depuis un autre type, ce qui permet de convertir une chaîne en entier.

```
>>> minuscule.capitalize() # La première lettre en majuscule
'Une chaine en minuscules'
>>> espaces = "   une   chaine avec  des espaces    "
>>> espaces.strip() # On retire les espaces au début et à la
    fin de la chaîne
'une   chaine avec  des espaces'
>>> titre = "introduction"
>>> titre.upper().center(20)
'    INTRODUCTION    '
>>>
```

La dernière instruction mérite quelques explications.

On appelle d'abord la méthode `upper` de l'objet `titre`. Cette méthode, comme vous l'avez vu plus haut, renvoie en majuscules la chaîne de caractères contenue dans l'objet.

On appelle ensuite la méthode `center`, méthode que nous n'avons pas encore vue et qui permet de centrer une chaîne. On lui passe en paramètre la taille de la chaîne que l'on souhaite obtenir. La méthode va ajouter alternativement un espace au début et à la fin de la chaîne, jusqu'à obtenir la longueur demandée. Dans cet exemple, `titre` contient la chaîne `'introduction'`, chaîne qui (en minuscules ou en majuscules) mesure 12 caractères. On demande à `center` de centrer cette chaîne dans un espace de 20 caractères. La méthode `center` va donc placer 4 espaces avant le titre et 4 espaces après, pour faire 20 caractères en tout.

Bon, mais maintenant, sur quel objet travaille `center` ? Sur `titre` ? Non. Sur la chaîne renvoyée par `titre.upper()`, c'est-à-dire le titre en majuscules. C'est pourquoi on peut « chaîner » ces deux méthodes : `upper`, comme la plupart des méthodes de chaînes, travaille sur une chaîne et renvoie une chaîne... qui elle aussi va posséder les méthodes propres à une chaîne de caractères. Si ce n'est pas très clair, faites quelques tests, avec `titre.upper()` et `titre.center(20)`, en passant par une seconde variable si nécessaire, pour vous rendre compte du mécanisme ; ce n'est pas bien compliqué.

Je n'ai mis ici que quelques méthodes, il y en a bien d'autres. Vous pouvez en voir la liste dans l'aide, en tapant, dans l'interpréteur : `help(str)`.

Formater et afficher une chaîne



Attends, on a appris à faire cela depuis cinq bons chapitres ! On ne va pas tout réapprendre quand même ?

Heureusement que non ! Mais nous allons apprendre à considérer ce que nous savons à travers le modèle objet. Et vous allez vous rendre compte que, la plupart du temps, nous n'avons fait qu'effleurer les fonctionnalités du langage.

Je ne vais pas revenir sur ce que j'ai dit, pour afficher une chaîne, on passe par la fonction `print`.

```
chaine = "Bonjour tout le monde !"
```

```
print(chaine)
```

Rien de nouveau ici. En revanche, nous allons un peu changer nos habitudes en ce qui concerne l'affichage de plusieurs variables.

Jusqu'ici, nous avons utilisé `print` en lui imputant plusieurs paramètres. Cela fonctionne mais nous allons voir une méthode légèrement plus souple, qui d'ailleurs n'est pas seulement utile pour l'affichage.

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom
    , nom, age))
Je m'appelle Paul Dupont et j'ai 21 ans.
```



Mais ! C'est quoi cela ?

Question légitime. Voyons un peu.

Première syntaxe de la méthode `format`

Nous avons utilisé une méthode de la classe `str` pour formater notre chaîne. De gauche à droite, nous avons :

- une chaîne de caractères qui ne présente rien de particulier, sauf ces accolades entourant des nombres, d'abord `0`, puis `1`, puis `2` ;
- nous appelons la méthode `format` de cette chaîne en lui passant en paramètres les variables à afficher, dans un ordre bien précis ;
- quand Python exécute cette méthode, il remplace dans notre chaîne `{0}` par la première variable passée à la méthode `format` (soit le prénom), `{1}` par la deuxième variable... et ainsi de suite.



Souvenez-vous qu'en programmation, on commence à compter à partir de 0.



Bien, mais on aurait pu faire exactement la même chose en passant plusieurs valeurs à `print`, non ?

Absolument. Mais rappelez-vous que cette fonctionnalité est bien plus puissante qu'un simple affichage, vous pouvez formater des chaînes de cette façon. Ici, nous avons directement affiché la chaîne formatée, mais nous aurions pu la stocker :

```
>>> nouvelle_chaine = "Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age)
>>>
```

Pour faire la même chose sans utiliser `format`, on aurait dû concaténer des chaînes, c'est-à-dire les mettre bout à bout en respectant une certaine syntaxe. Nous allons voir cela un peu plus loin mais cette solution reste plus élégante.

Dans cet exemple, nous avons appelé les variables dans l'ordre où nous les placions dans `format`, mais ce n'est pas une obligation. Considérez cet exemple :

```
>>> prenom = "Paul"
>>> nom = "Dupont"
>>> age = 21
>>> print( \
...     "Je m'appelle {0} {1} ({3} {0} pour l'administration) et j'ai {2} " \
...     "ans.".format(prenom, nom, age, nom.upper()))
Je m'appelle Paul Dupont (DUPONT Paul pour l'administration) et j'ai 21 ans.
```

J'ai coupé notre instruction, plutôt longue, à l'aide du signe « \ » placé avant un saut de ligne, pour indiquer à Python que l'instruction se prolongeait au-dessous.

Si vous avez du mal à comprendre l'exemple, relisez l'instruction en remplaçant vous-mêmes les nombres entre accolades par les variables.



Dans la plupart des cas, on ne précise pas le numéro de la variable entre accolades.

```
>>> date = "Dimanche 24 juillet 2011"
>>> heure = "17:00"
>>> print("Cela s'est produit le {}, à {}.".format(date, heure))
Cela s'est produit le Dimanche 24 juillet 2011, à 17:00.
>>>
```

Naturellement, cela ne fonctionne que si vous donnez les variables dans le bon ordre dans `format`.

Cette syntaxe suffit la plupart du temps mais elle n'est pas forcément intuitive quand on insère beaucoup de variables : on doit retenir leur position dans l'appel à `format` pour comprendre laquelle est affichée à tel endroit. Mais il existe une autre syntaxe.

Seconde syntaxe de la méthode `format`

On peut également nommer les variables que l'on va afficher, c'est souvent plus intuitif que d'utiliser leur indice. Voici un nouvel exemple :

```
# formatage d'une adresse
adresse = """
  {no_rue}, {nom_rue}
  {code_postal} {nom_ville} ({pays})"""
  .format(no_rue=5, nom_rue="rue des Postes", code_postal=75003
      , nom_ville="Paris", pays="France")
print(adresse)
```

... affichera :

```
5, rue des Postes
  75003 Paris (France)
```

Je pense que vous voyez assez précisément en quoi consiste cette deuxième syntaxe de `format`. Au lieu de donner des nombres entre accolades, on spécifie des noms de variables qui doivent correspondre à ceux fournis comme mots-clés dans la méthode `format`. Je ne m'attarderai pas davantage sur ce point, je pense qu'il est assez clair comme cela.

La concaténation de chaînes

Nous allons glisser très rapidement sur le concept de concaténation, assez intuitif d'ailleurs. On cherche à regrouper deux chaînes en une, en mettant la seconde à la suite de la première. Cela se fait le plus simplement du monde :

```
>>> prenom = "Paul"
>>> message = "Bonjour"
>>> chaine_complete = message + prenom # On utilise le symbole
    '+' pour concaténer deux chaînes
... print(chaine_complete) # Résultat :
BonjourPaul
>>> # Pas encore parfait, il manque un espace
... # Qu'à cela ne tienne !
... chaine_complete = message + " " + prenom
>>> print(chaine_complete) # Résultat :
Bonjour Paul
>>>
```

C'est assez clair je pense. Le signe « + » utilisé pour ajouter des nombres est ici utilisé pour **concaténer** deux chaînes. Essayons à présent de concaténer des chaînes et des nombres :

```
>>> age = 21
>>> message = "J'ai " + age + " ans."
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Python se fâche tout rouge ! Certains langages auraient accepté cette syntaxe sans sourciller mais Python n'aime pas cela du tout.

Au début de la première partie, nous avons dit que Python était un langage à **typage dynamique**, ce qui signifie qu'il identifie lui-même les types de données et que les variables peuvent changer de type au cours du programme. Mais Python est aussi un langage **fortement typé**, et cela veut dire que les types de données ne sont pas là juste pour faire joli, on ne peut pas les ignorer. Ainsi, on veut ici ajouter une chaîne à un entier et à une autre chaîne. Python ne comprend pas : est-ce que les chaînes contiennent des nombres qu'il doit convertir pour les ajouter à l'entier ou est-ce que l'entier doit être converti en chaîne puis concaténé avec les autres chaînes ? Python ne sait pas. Il ne le fera pas tout seul. Mais il se révèle être de bonne volonté puisqu'il suffit de lui demander de convertir l'entier pour pouvoir le concaténer aux autres chaînes.

```
>>> age = 21
>>> message = "J'ai " + str(age) + " ans."
>>> print(message)
J'ai 21 ans.
>>>
```

On appelle `str` pour convertir un objet en une chaîne de caractères, comme nous avons appelé `int` pour convertir un objet en entier. C'est le même mécanisme, sauf que convertir un entier en chaîne de caractères ne lèvera vraissemblablement aucune exception.

Le typage fort de Python est important, il est un fondement de sa philosophie. J'ai tendance à considérer qu'un langage faiblement typé crée des erreurs qui sont plus difficiles à repérer alors qu'ici, il nous suffit de convertir explicitement le type pour que Python sache ce qu'il doit faire.

Parcours et sélection de chaînes

Nous avons vu très rapidement dans la première partie un moyen de parcourir des chaînes. Nous allons en voir ici un second qui fonctionne par indice.

Parcours par indice

Vous devez vous en souvenir : j'ai dit qu'une chaîne de caractères était une séquence constituée... de caractères. En fait, une chaîne de caractères est elle-même constituée de chaînes de caractères, chacune d'elles n'étant composée que d'un seul caractère.

Accéder aux caractères d'une chaîne

Nous allons apprendre à accéder aux lettres constituant une chaîne. Par exemple, nous souhaitons sélectionner la première lettre d'une chaîne.

```
>>> chaine = "Salut les ZEROS !"
>>> chaine[0] # Première lettre de la chaîne
'S'
>>> chaine[2] # Troisième lettre de la chaîne
'l'
>>> chaine[-1] # Dernière lettre de la chaîne
'!'
>>>
```

On précise entre crochets `[]` l'indice (la position du caractère auquel on souhaite accéder).

Rappelez-vous, on commence à compter à partir de 0. La première lettre est donc à l'indice 0, la deuxième à l'indice 1, la troisième à l'indice 2... On peut accéder aux lettres en partant de la fin à l'aide d'un indice négatif. Quand vous tapez `chaine[-1]`, vous accédez ainsi à la dernière lettre de la chaîne (enfin, au dernier caractère, qui n'est pas une lettre ici).

On peut obtenir la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction `len`.

```
>>> chaine = "Salut"
>>> len(chaine)
5
>>>
```



Pourquoi ne pas avoir défini cette fonction comme une méthode de la classe `str` ? Pourquoi ne pourrait-on pas faire `chaine.len()` ?

En fait c'est un peu le cas, mais nous le verrons bien plus loin. `str` n'est qu'un exemple parmi d'autres de séquences (on en découvrira d'autres dans les prochains chapitres) et donc les développeurs de Python ont préféré créer une fonction qui travaillerait sur les séquences au sens large, plutôt qu'une méthode pour chacune de ces classes.

Méthode de parcours par `while`

Vous en savez assez pour parcourir une chaîne grâce à la boucle `while`. Notez que, dans la plupart des cas, on préférera parcourir une séquence avec `for`. Il est néanmoins bon de savoir procéder de différentes manières, cela vous sera utile parfois.

Voici le code auquel vous pourriez arriver :

```
chaine = "Salut"
```

```
i = 0 # On appelle l'indice 'i' par convention
while i < len(chaine):
    print(chaine[i]) # On affiche le caractère à chaque tour de
        boucle
    i += 1
```

N'oubliez pas d'incrémenter `i`, sinon vous allez avoir quelques surprises.

Si vous essayez d'accéder à un indice qui n'existe pas (par exemple 25 alors que votre chaîne ne fait que 20 caractères de longueur), Python lèvera une exception de type `IndexError`.

Enfin, une dernière petite chose : vous ne pouvez changer les lettres de la chaîne en utilisant les indices.

```
>>> mot = "lac"
>>> mot[0] = "b" # On veut remplacer 'l' par 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python n'est pas content. Il ne veut pas que vous utilisiez les indices pour modifier des caractères de la chaîne. Pour ce faire, il va falloir utiliser la sélection.

Sélection de chaînes

Nous allons voir comment sélectionner une partie de la chaîne. Si je souhaite, par exemple, sélectionner les deux premières lettres de la chaîne, je procéderai comme dans l'exemple ci-dessous.

```
>>> presentation = "salut"
>>> presentation[0:2] # On sélectionne les deux premières
   lettres
'sa'
>>> presentation[2:len(presentation)] # On sélectionne la chaî
   ne sauf les deux premières lettres
'lut'
>>>
```

La sélection consiste donc à extraire une partie de la chaîne. Cette opération renvoie le morceau de la chaîne sélectionné, sans modifier la chaîne d'origine.

Sachez que l'on peut sélectionner du début de la chaîne jusqu'à un indice, ou d'un indice jusqu'à la fin de la chaîne, sans préciser autant d'informations que dans nos exemples. Python comprend très bien si on sous-entend certaines informations.

```
>>> presentation[:2] # Du début jusqu'à la troisième lettre non
    comprise
'sa'
```

```
>>> presentation[2:] # De la troisième lettre (comprise) à la
    fin
'lut'
>>>
```

Maintenant, nous pouvons reprendre notre exemple de tout à l'heure pour constituer une nouvelle chaîne, en remplaçant une lettre par une autre :

```
>>> mot = "lac"
>>> mot = "b" + mot[1:]
>>> print(mot)
bac
>>>
```

Voilà !

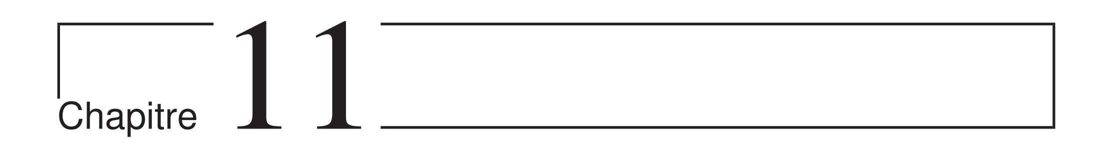


Cela reste assez peu intuitif, non ?

Pour remplacer des lettres, cela paraît un peu lourd en effet. Et d'ailleurs on s'en sert assez rarement pour cela. Pour rechercher/remplacer, nous avons à notre disposition les méthodes `count`, `find` et `replace`, à savoir « compter », « rechercher » et « remplacer ».

En résumé

- Les variables utilisées jusqu'ici sont en réalité des objets.
- Les types de données utilisés jusqu'ici sont en fait des classes. Chaque objet est modelé sur une classe.
- Chaque classe définit certaines fonctions, appelées méthodes, qui seront accessibles depuis l'objet grâce à `objet.methode(arguments)`.
- On peut directement accéder à un caractère d'une chaîne grâce au code suivant : `chaine[position_dans_la_chaine]`.
- Il est tout à fait possible de sélectionner une partie de la chaîne grâce au code suivant : `chaine[indice_debut:indice_fin]`.

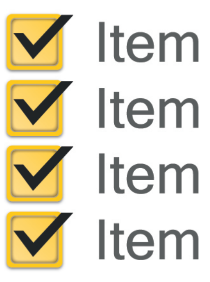


Les listes et tuples (1/2)

Difficulté : 

J'aurai réussi à vous faire connaître et, j'espère, aimer le langage Python sans vous apprendre les listes. Mais allons ! Cette époque est révolue. Maintenant que nous commençons à étudier l'objet sous toutes ses formes, je ne vais pas pouvoir garder le secret plus longtemps : il existe des listes en Python. Pour ceux qui ne voient même pas de quoi je parle, vous allez vite vous rendre compte qu'avec les dictionnaires (que nous verrons plus loin), c'est un type, ou plutôt une classe, dont on aura du mal à se passer.

Les listes sont des séquences. En fait, leur nom est plutôt explicite, puisque ce sont des objets capables de contenir d'autres objets de n'importe quel type. On peut avoir une liste contenant plusieurs nombres entiers (1, 2, 50, 2000 ou plus, peu importe), une liste contenant des flottants, une liste contenant des chaînes de caractères... et une liste mélangeant ces objets de différents types.



Créons et éditons nos premières listes

D'abord c'est quoi, une liste ?

En Python, les listes sont des objets qui peuvent en contenir d'autres. Ce sont donc des séquences, comme les chaînes de caractères, mais au lieu de contenir des caractères, elles peuvent contenir n'importe quel objet. Comme d'habitude, on va s'occuper du concept des listes avant de voir tout son intérêt.

Création de listes

On a deux moyens de créer des listes. Si je vous dis que la classe d'une liste s'appelle, assez logiquement, `list`, vous devriez déjà voir une manière de créer une liste.

Non ? . . .

Vous allez vous habituer à cette syntaxe :

```
>>> ma_liste = list() # On crée une liste vide
>>> type(ma_liste)
<class 'list'>
>>> ma_liste
[]
>>>
```

Là encore, on utilise le nom de la classe comme une fonction pour **instancier** un objet de cette classe.

Quand vous affichez la liste, vous pouvez constater qu'elle est vide. Entre les crochets (qui sont les délimiteurs des listes en Python), il n'y a rien. On peut également utiliser ces crochets pour créer une liste.

```
>>> ma_liste = [] # On crée une liste vide
>>>
```

Cela revient au même, vous pouvez vérifier. Toutefois, on peut également créer une liste non vide, en lui indiquant directement à la création les objets qu'elle doit contenir.

```
>>> ma_liste = [1, 2, 3, 4, 5] # Une liste avec cinq objets
>>> print(ma_liste)
[1, 2, 3, 4, 5]
>>>
```

La liste que nous venons de créer compte cinq objets de type `int`. Ils sont classés par ordre croissant. Mais rien de tout cela n'est obligatoire.

- Vous pouvez faire des listes de toute longueur.
- Les listes peuvent contenir n'importe quel type d'objet.

– Les objets dans une liste peuvent être mis dans un ordre quelconque. Toutefois, la structure d'une liste fait que chaque objet *a sa place* et que l'ordre compte.

```
>>> ma_liste = [1, 3.5, "une chaine", []]
>>>
```

Nous avons créé ici une liste contenant quatre objets de types différents : un entier, un flottant, une chaîne de caractères et... une autre liste.

Voyons à présent comment accéder aux éléments d'une liste :

```
>>> ma_liste = ['c', 'f', 'm']
>>> ma_liste[0] # On accède au premier élément de la liste
'c'
>>> ma_liste[2] # Troisième élément
'm'
>>> ma_liste[1] = 'Z' # On remplace 'f' par 'Z'
>>> ma_liste
['c', 'Z', 'm']
>>>
```

Comme vous pouvez le voir, on accède aux éléments d'une liste de la même façon qu'on accède aux caractères d'une chaîne de caractères : on indique entre crochets l'indice de l'élément qui nous intéresse.

Contrairement à la classe `str`, la classe `list` vous permet de remplacer un élément par un autre. Les listes sont en effet des types dits **mutables**.

Insérer des objets dans une liste

On dispose de plusieurs méthodes, définies dans la classe `list`, pour ajouter des éléments dans une liste.

Ajouter un élément à la fin de la liste

On utilise la méthode `append` pour ajouter un élément à la fin d'une liste.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste.append(56) # On ajoute 56 à la fin de la liste
>>> ma_liste
[1, 2, 3, 56]
>>>
```

C'est assez simple non ? On passe en paramètre de la méthode `append` l'objet que l'on souhaite ajouter à la fin de la liste.



La méthode append, comme beaucoup de méthodes de listes, travaille directement sur l'objet et ne renvoie donc rien !

Ceci est extrêmement important. Dans le chapitre précédent, nous avons vu qu'aucune des méthodes de chaînes ne modifie l'objet d'origine mais qu'elles renvoient toutes un nouvel objet, qui est la chaîne modifiée. Ici c'est le contraire : les méthodes de listes ne renvoient rien mais modifient l'objet d'origine. Regardez ce code si ce n'est pas bien clair :

```
>>> chaine1 = "une petite phrase"
>>> chaine2 = chaine1.upper() # On met en majuscules chaine1
>>> chaine1                   # On affiche la chaîne d'origine
'une petite phrase'
>>> # Elle n'a pas été modifiée par la méthode upper
... chaine2                   # On affiche chaine2
'UNE PETITE PHRASE'
>>> # C'est chaine2 qui contient la chaîne en majuscules
... # Voyons pour les listes à présent
... liste1 = [1, 5.5, 18]
>>> liste2 = liste1.append(-15) # On ajoute -15 à liste1
>>> liste1                      # On affiche liste1
[1, 5.5, 18, -15]
>>> # Cette fois, l'appel de la méthode a modifié l'objet d'origine (liste1)
... # Voyons ce que contient liste2
... liste2
>>> # Rien ? Vérifions avec print
... print(liste2)
None
>>>
```

Je vais expliquer les dernières lignes. Mais d'abord, il faut que vous fassiez bien la différence entre les méthodes de chaînes, où l'objet d'origine n'est jamais modifié et qui renvoient un nouvel objet, et les méthodes de listes, qui ne renvoient rien mais modifient l'objet d'origine.

J'ai dit que les méthodes de listes ne renvoient rien. On va pourtant essayer de « capturer » la valeur de retour dans `liste2`. Quand on essaye d'afficher la valeur de `liste2` par saisie directe, on n'obtient rien. Il faut l'afficher avec `print` pour savoir ce qu'elle contient : `None`. C'est l'objet vide de Python. En réalité, quand une fonction ne renvoie rien, elle renvoie `None`. Vous retrouverez peut-être cette valeur de temps à autre, ne soyez donc pas surpris.

Insérer un élément dans la liste

Nous allons passer assez rapidement sur cette seconde méthode. On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu. On utilise pour cela la méthode `insert`.

```
>>> ma_liste = ['a', 'b', 'd', 'e']
>>> ma_liste.insert(2, 'c') # On insère 'c' à l'indice 2
>>> print(ma_liste)
```

```
['a', 'b', 'c', 'd', 'e']
```

Quand on demande d'insérer `c` à l'indice 2, la méthode va décaler les objets d'indice supérieur ou égal à 2. `c` va donc s'intercaler entre `b` et `d`.

Concaténation de listes

On peut également agrandir des listes en les concaténant avec d'autres.

```
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste2 = [8, 9, 10]
>>> ma_liste1.extend(ma_liste2) # On insère ma_liste2 à la fin
    de ma_liste1
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 = [3, 4, 5]
>>> ma_liste1 + ma_liste2
[3, 4, 5, 8, 9, 10]
>>> ma_liste1 += ma_liste2 # Identique à extend
>>> print(ma_liste1)
[3, 4, 5, 8, 9, 10]
>>>
```

Voici les différentes façons de concaténer des listes. Vous pouvez remarquer l'opérateur `+` qui concatène deux listes entre elles et renvoie le résultat. On peut utiliser `+=` assez logiquement pour étendre une liste. Cette façon de faire revient au même qu'utiliser la méthode `extend`.

Suppression d'éléments d'une liste

Nous allons voir rapidement comment supprimer des éléments d'une liste, avant d'apprendre à les parcourir. Vous allez vite pouvoir constater que cela se fait assez simplement. Nous allons voir deux méthodes pour supprimer des éléments d'une liste :

- le mot-clé `del` ;
- la méthode `remove`.

Le mot-clé `del`

C'est un des mots-clés de Python, que j'aurais pu vous montrer plus tôt. Mais les applications de `del` me semblaient assez peu pratiques avant d'aborder les listes.

`del` (abréviation de *delete*) signifie « supprimer » en anglais. Son utilisation est des plus simple : `del variable_a_supprimer`. Voyons un exemple.

```
>>> variable = 34
>>> variable
```

```
34
>>> del variable
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
>>>
```

Comme vous le voyez, après l'utilisation de `del`, la variable n'existe plus. Python l'efface tout simplement. Mais on peut également utiliser `del` pour supprimer des éléments d'une séquence, comme une liste, et c'est ce qui nous intéresse ici.

```
>>> ma_liste = [-5, -2, 1, 4, 7, 10]
>>> del ma_liste[0] # On supprime le premier élément de la
    liste
>>> ma_liste
[-2, 1, 4, 7, 10]
>>> del ma_liste[2] # On supprime le troisième élément de la
    liste
>>> ma_liste
[-2, 1, 7, 10]
>>>
```

La méthode `remove`

On peut aussi supprimer des éléments de la liste grâce à la méthode `remove` qui prend en paramètre non pas l'indice de l'élément à supprimer, mais l'élément lui-même.

```
>>> ma_liste = [31, 32, 33, 34, 35]
>>> ma_liste.remove(32)
>>> ma_liste
[31, 33, 34, 35]
>>>
```

La méthode `remove` parcourt la liste et en retire l'élément que vous lui passez en paramètre. C'est une façon de faire un peu différente et vous appliquerez `del` ou `remove` en fonction de la situation.



La méthode `remove` ne retire que la première occurrence de la valeur trouvée dans la liste !

Notez au passage que le mot-clé `del` n'est pas une méthode de liste. Il s'agit d'une fonctionnalité de Python qu'on retrouve dans la plupart des objets conteneurs, tels que les listes que nous venons de voir, ou les dictionnaires que nous verrons plus tard. D'ailleurs, `del` sert plus généralement à supprimer non seulement des éléments d'une séquence mais aussi, comme nous l'avons vu, des variables.

Nous allons à présent voir comment parcourir une liste, même si vous devez déjà avoir votre petite idée sur la question.

Le parcours de listes

Vous avez déjà dû vous faire une idée des méthodes pour parcourir une liste. Je vais passer brièvement dessus, vous ne verrez rien de nouveau ni, je l'espère, de très surprenant.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> i = 0 # Notre indice pour la boucle while
>>> while i < len(ma_liste):
...     print(ma_liste[i])
...     i += 1 # On incrémente i, ne pas oublier !
...
a
b
c
d
e
f
g
h
>>> # Cette méthode est cependant préférable
... for elt in ma_liste: # elt va prendre les valeurs successives des éléments de ma_liste
...     print(elt)
...
a
b
c
d
e
f
g
h
>>>
```

Il s'agit des mêmes méthodes de parcours que nous avons vues pour les chaînes de caractères, au chapitre précédent. Nous allons cependant aller un peu plus loin.

La fonction `enumerate`

Les deux méthodes que nous venons de voir possèdent toutes deux des inconvénients :

– la méthode utilisant `while` est plus longue à écrire, moins intuitive et elle est perméable aux boucles infinies, si l'on oublie d'incrémenter la variable servant de comp-

teur ;
– la méthode par `for` se contente de parcourir la liste en capturant les éléments dans une variable, sans qu'on puisse savoir où ils sont dans la liste.

C'est vrai dans le cas que nous venons de voir. Certains codeurs vont combiner les deux méthodes pour plus de flexibilité mais, très souvent, le code obtenu est moins lisible. Heureusement, les développeurs de Python ont pensé à nous.

```
>>> ma_liste = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> for i, elt in enumerate(ma_liste):
...     print("À l'indice {} se trouve {}.".format(i, elt))
...
À l'indice 0 se trouve a.
À l'indice 1 se trouve b.
À l'indice 2 se trouve c.
À l'indice 3 se trouve d.
À l'indice 4 se trouve e.
À l'indice 5 se trouve f.
À l'indice 6 se trouve g.
À l'indice 7 se trouve h.
>>>
```

Pas de panique !

Nous avons ici une boucle `for` un peu surprenante. Entre `for` et `in`, nous avons deux variables, séparées par une virgule.

En fait, `enumerate` prend en paramètre une liste et renvoie un objet qui peut être associé à une liste contenant deux valeurs par élément : l'indice et l'élément de la liste parcouru.

Ce n'est sans doute pas encore très clair. Essayons d'afficher cela un peu mieux :

```
>>> for elt in enumerate(ma_liste):
...     print(elt)
...
(0, 'a')
(1, 'b')
(2, 'c')
(3, 'd')
(4, 'e')
(5, 'f')
(6, 'g')
(7, 'h')
>>>
```

Quand on parcourt chaque élément de l'objet renvoyé par `enumerate`, on voit des **tuples** qui contiennent deux éléments : d'abord l'indice, puis l'objet se trouvant à cet indice dans la liste passée en argument à la fonction `enumerate`.



Les tuples sont des séquences, assez semblables aux listes, sauf qu'on ne peut modifier un tuple après qu'il ait été créé. Cela signifie qu'on définit le contenu d'un tuple (les objets qu'il doit contenir) lors de sa création, mais qu'on ne peut en ajouter ou en retirer par la suite.

Si les parenthèses vous déconcertent trop, vous pouvez imaginer, à la place, des crochets : dans cet exemple, cela revient au même.

Quand on utilise `enumerate`, on capture l'indice et l'élément dans deux variables distinctes. Voyons un autre exemple pour comprendre ce mécanisme :

```
>>> autre_liste = [
...     [1, 'a'],
...     [4, 'd'],
...     [7, 'g'],
...     [26, 'z'],
... ] # J'ai étalé la liste sur plusieurs lignes
>>> for nb, lettre in autre_liste:
...     print("La lettre {} est la {}e de l'alphabet.".format(
    lettre, nb))
...
La lettre a est la 1e de l'alphabet.
La lettre d est la 4e de l'alphabet.
La lettre g est la 7e de l'alphabet.
La lettre z est la 26e de l'alphabet.
>>>
```

J'espère que c'est assez clair dans votre esprit. Dans le cas contraire, décomposez ces exemples, le déclic devrait se faire.



On écrit ici la définition de la liste sur plusieurs lignes pour des raisons de lisibilité. On n'est pas obligé de mettre des anti-slashs « \ » en fin de ligne car, tant que Python ne trouve pas de crochet fermant la liste, il continue d'attendre sans interpréter la ligne. Vous pouvez d'ailleurs le constater avec les points qui remplacent les chevrons au début de la ligne, tant que la liste n'a pas été refermée.



Quand on travaille sur une liste que l'on parcourt en même temps, on peut se retrouver face à des erreurs assez étranges, qui paraissent souvent incompréhensibles au début.

Par exemple, on peut être confronté à des exceptions `IndexError` si on tente de supprimer certains éléments d'une liste en la parcourant.

Nous verrons au prochain chapitre comment faire cela proprement, pour l'heure il vous suffit de vous méfier d'un parcours qui modifie une liste, surtout sa structure. D'une façon générale, évitez de parcourir une liste dont la taille évolue en même temps.

Allez ! On va jeter un coup d'œil aux tuples, pour conclure ce chapitre !

Un petit coup d'œil aux tuples

Nous avons brièvement vu les **tuples** un peu plus haut, grâce à la fonction `enumerate`. Les tuples sont des listes immuables, qu'on ne peut modifier. En fait, vous allez vous rendre compte que nous utilisons depuis longtemps des tuples sans nous en rendre compte.

Un tuple se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets.

```
tuple_vide = ()
tuple_non_vide = (1,)
tuple_non_vide = (1, 3, 5)
```

À la différence des listes, les tuples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

Une petite subtilité ici : si on veut créer un tuple contenant un unique élément, on doit quand même mettre une virgule après celui-ci. Sinon, Python va automatiquement supprimer les parenthèses et on se retrouvera avec une variable lambda et non un tuple contenant cette variable.



Mais à quoi cela sert-il ?

Il est assez rare que l'on travaille directement sur des tuples. C'est, après tout, un type que l'on ne peut pas modifier. On ne peut supprimer d'éléments d'un tuple, ni en ajouter. Cela vous paraît peut-être encore assez abstrait mais il peut être utile de travailler sur des données sans pouvoir les modifier.

En attendant, voyons plutôt les cas où nous avons utilisé des tuples sans le savoir.

Affectation multiple

Tous les cas que nous allons voir sont des cas d'affectation multiple. Vous vous souvenez ?

```
>>> a, b = 3, 4
>>> a
3
>>> b
4
>>>
```

On a également utilisé cette syntaxe pour permuter deux variables. Eh bien, cette syntaxe passe par des tuples qui ne sont pas déclarés explicitement. Vous pourriez écrire :

```
>>> (a, b) = (3, 4)
>>>
```

Quand Python trouve plusieurs variables ou valeurs séparées par des virgules et sans délimiteur, il va les mettre dans des tuples. Dans le premier exemple, les parenthèses sont sous-entendues et Python comprend ce qu'il doit faire.

Une fonction renvoyant plusieurs valeurs

Nous ne l'avons pas vu jusqu'ici mais une fonction peut renvoyer deux valeurs ou même plus :

```
def decomposer(entier, divise_par):
    """Cette fonction retourne la partie entière et le reste de
    entier / divise_par"""

    p_e = entier // divise_par
    reste = entier % divise_par
    return p_e, reste
```

Et on peut ensuite capturer la partie entière et le reste dans deux variables, au retour de la fonction :

```
>>> partie_entiere, reste = decomposer(20, 3)
>>> partie_entiere
6
>>> reste
2
>>>
```

Là encore, on passe par des tuples sans que ce soit indiqué explicitement à Python. Si vous essayez de faire `retour = decomposer(20, 3)`, vous allez capturer un tuple contenant deux éléments : la partie entière et le reste de 20 divisé par 3.

Nous verrons plus loin d'autres exemples de tuples et d'autres utilisations. Je pense que cela suffit pour cette fois.

En résumé

- Une liste est une séquence mutable pouvant contenir plusieurs autres objets.
- Une liste se construit ainsi : `liste = [element1, element2, elementN]`.
- On peut insérer des éléments dans une liste à l'aide des méthodes `append`, `insert` et `extends`.

- On peut supprimer des éléments d'une liste grâce au mot-clé `del` ou à la méthode `remove`.
- Un tuple est une séquence pouvant contenir des objets. À la différence de la liste, le tuple ne peut être modifié une fois créé.

Chapitre 12

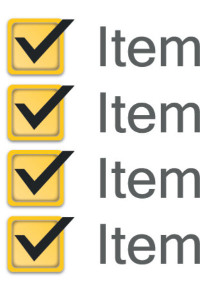
Les listes et tuples (2/2)

Difficulté : 

Les listes sont très utilisées en Python. Elles sont liées à pas mal de fonctionnalités, dont certaines plutôt complexes. Aussi ai-je préféré scinder l'approche des listes en deux chapitres. Vous allez voir dans celui-ci quelques fonctionnalités qui ne s'appliquent qu'aux listes et aux tuples, et qui pourront vous être extrêmement utiles. Je vous conseille donc, avant tout, d'être bien à l'aise avec les listes et leur création, parcours, édition, suppression...

D'autre part, comme pour la plupart des sujets abordés, je ne peux faire un tour d'horizon exhaustif de toutes les fonctionnalités de chaque objet présenté. Je vous invite donc à lire la documentation, en tapant `help(list)`, pour accéder à une liste exhaustive des méthodes.

C'est parti !



Entre chaînes et listes

Nous allons voir un moyen de transformer des chaînes en listes et réciproquement.

Il est assez surprenant, de prime abord, qu'une conversion soit possible entre ces deux types qui sont tout de même assez différents. Mais comme on va le voir, il ne s'agit pas réellement d'une conversion. Il va être difficile de démontrer l'utilité de cette fonctionnalité tout de suite, mieux valent quelques exemples.

Des chaînes aux listes

Pour « convertir » une chaîne en liste, on va utiliser une méthode de chaîne nommée `split` (« éclater » en anglais). Cette méthode prend un paramètre qui est une autre chaîne, souvent d'un seul caractère, définissant comment on va découper notre chaîne initiale.

C'est un peu compliqué et cela paraît très tordu… mais regardez plutôt :

```
>>> ma_chaine = "Bonjour à tous"
>>> ma_chaine.split(" ")
['Bonjour', 'à', 'tous']
>>>
```

On passe en paramètre de la méthode `split` une chaîne contenant un unique espace. La méthode renvoie une liste contenant les trois mots de notre petite phrase. Chaque mot se trouve dans une case de la liste.

C'est assez simple en fait : quand on appelle la méthode `split`, celle-ci découpe la chaîne en fonction du paramètre donné. Ici la première case de la liste va donc du début de la chaîne au premier espace (non inclus), la deuxième case va du premier espace au second, et ainsi de suite jusqu'à la fin de la chaîne.

Sachez que `split` possède un paramètre par défaut, un code qui représente les espaces, les tabulations et les sauts de ligne. Donc vous pouvez très bien faire `ma_chaine.split()`, cela revient ici au même.

Des listes aux chaînes

Voyons l'inverse à présent, c'est-à-dire si on a une liste contenant plusieurs chaînes de caractères que l'on souhaite fusionner en une seule. On utilise la méthode de chaîne `join` (« joindre » en anglais). Sa syntaxe est un peu surprenante :

```
>>> ma_liste = ['Bonjour', 'à', 'tous']
>>> " ".join(ma_liste)
'Bonjour à tous'
>>>
```

En paramètre de la méthode `join`, on passe la liste des chaînes que l'on souhaite « ressouder ». La méthode va travailler sur l'objet qui l'appelle, ici une chaîne de caractères contenant un unique espace. Elle va insérer cette chaîne entre chaque paire de chaînes de la liste, ce qui au final nous donne la chaîne de départ, « Bonjour à tous ».



N'aurait-il pas été plus simple ou plus logique de faire une méthode de liste, prenant en paramètre la chaîne faisant la jonction ?

Ce choix est en effet contesté mais, pour ma part, je ne trancherai pas. Le fait est que c'est cette méthode qui a été choisie et, avec un peu d'habitude, on arrive à bien lire le résultat obtenu. D'ailleurs, nous allons voir comment appliquer concrètement ces deux méthodes.

Une application pratique

Admettons que nous ayons un nombre flottant dont nous souhaitons afficher la partie entière et les trois premières décimales uniquement de la partie flottante. Autrement dit, si on a un nombre flottant tel que « 3.999999999999998 », on souhaite obtenir comme résultat « 3.999 ». D'ailleurs, ce serait plus joli si on remplaçait le point décimal par la virgule, à laquelle nous sommes plus habitués.

Là encore, je vous invite à essayer de faire ce petit exercice par vous-même. On part du principe que la valeur de retour de la fonction chargée de la pseudo-conversion est une chaîne de caractères. Voici quelques exemples d'utilisation de la fonction que vous devriez coder :

```
>>> afficher_flottant(3.99999999999998)
'3,999'
>>> afficher_flottant(1.5)
'1,5'
>>>
```

Voici la correction que je vous propose :

```
def afficher_flottant(flottant):
    """Fonction prenant en paramètre un flottant et renvoyant
        une chaîne de caractères représentant la troncature de
        ce nombre. La partie flottante doit avoir une longueur
        maximum de 3 caractères.

    De plus, on va remplacer le point décimal par la virgule"""

    if type(flottant) is not float:
        raise TypeError("Le paramètre attendu doit être un
            flottant")
    flottant = str(flottant)
    partie_entiere, partie_flottante = flottant.split(".")
```

```
    # La partie entière n'est pas à modifier
    # Seule la partie flottante doit être tronquée
    return ",".join([partie_entiere, partie_flottante[:3]])
```

En s'assurant que le type passé en paramètre est bien un flottant, on garantit qu'il n'y aura pas d'erreur lors du fractionnement de la chaîne. On est sûr qu'il y aura forcément une partie entière et une partie flottante séparées par un point, même si la partie flottante n'est constituée que d'un 0. Si vous n'y êtes pas arrivés par vous-même, étudiez bien cette solution, elle n'est pas forcément évidente au premier coup d'œil. On fait intervenir un certain nombre de mécanismes que vous avez vus il y a peu, tâchez de bien les comprendre.

Les listes et paramètres de fonctions

Nous allons droit vers une fonctionnalité des plus intéressantes, qui fait une partie de la puissance de Python. Nous allons étudier un cas assez particulier avant de généraliser : les fonctions dont le nombre de paramètres est inconnu.

Notez malgré tout que ce point est assez délicat. Si vous n'arrivez pas bien à le comprendre, laissez cette section de côté, cela ne vous pénalisera pas.

Les fonctions dont on ne connaît pas à l'avance le nombre de paramètres

Vous devriez tout de suite penser à la fonction `print` : on lui passe une liste de paramètres qu'elle va afficher, dans l'ordre où ils sont placés, séparés par un espace (ou tout autre délimiteur choisi).

Vous n'allez peut-être pas trouver d'applications de cette fonctionnalité dans l'immédiat mais, tôt ou tard, cela arrivera. La syntaxe est tellement simple que c'en est déconcertant :

```
def fonction(*parametres):
```

On place une étoile `*` devant le nom du paramètre qui accueillera la liste des arguments. Voyons plus précisément comment cela se présente :

```
>>> def fonction_inconnue(*parametres):
...     """Test d'une fonction pouvant être appelée avec un
    nombre variable de paramètres"""
...
...     print("J'ai reçu : {}.".format(parametres))
...
>>> fonction_inconnue() # On appelle la fonction sans paramètre
J'ai reçu : ().
>>> fonction_inconnue(33)
J'ai reçu : (33,).
>>> fonction_inconnue('a', 'e', 'f')
```

```
J'ai reçu : ('a', 'e', 'f').
>>> var = 3.5
>>> fonction_inconnue(var, [4], "...")
J'ai reçu : (3.5, [4], '...').
>>>
```

Je pense que cela suffit. Comme vous le voyez, on peut appeler la `fonction_inconnue` avec un nombre indéterminé de paramètres, allant de 0 à l'infini (enfin, théoriquement). Le fait de préciser une étoile `*` devant le nom du paramètre fait que Python va placer tous les paramètres de la fonction dans un **tuple**, que l'on peut ensuite traiter comme on le souhaite.



Et les paramètres nommés dans l'histoire ? Comment sont-ils insérés dans le tuple ?

Ils ne le sont pas. Si vous tapez `fonction_inconnue(couleur="rouge")`, vous allez avoir une erreur : `fonction_inconnue() got an unexpected keyword argument 'couleur'`. Nous verrons au prochain chapitre comment capturer ces paramètres nommés.

Vous pouvez bien entendu définir une fonction avec plusieurs paramètres qui doivent être fournis quoi qu'il arrive, suivis d'une liste de paramètres variables :

```
def fonction_inconnue(nom, prenom, *commentaires):
```

Dans cet exemple de définition de fonction, vous devez impérativement préciser un nom et un prénom, et ensuite vous mettez ce que vous voulez en commentaire, aucun paramètre, un, deux... ce que vous voulez.



Si on définit une liste variable de paramètres, elle doit se trouver après la liste des paramètres standard.

Au fond, cela est évident. Vous ne pouvez avoir une définition de fonction comme `def fonction_inconnue(*parametres, nom, prenom)`. En revanche, si vous souhaitez avoir des paramètres nommés, il faut les mettre après cette liste. Les paramètres nommés sont un peu une exception puisqu'ils ne figureront de toute façon pas dans le tuple obtenu. Voyons par exemple la définition de la fonction `print` :

```
print(value, ..., sep=' ', end='\n', file=sys.stdout)
```

Ne nous occupons pas du dernier paramètre. Il définit le descripteur vers lequel `print` envoie ses données ; par défaut, c'est l'écran.



D'où viennent ces points de suspension dans les paramètres ?

En fait, il s'agit d'un affichage un peu plus agréable. Si on veut réellement avoir la définition en code Python, on retombera plutôt sur :

```
def print(*values, sep=' ', end='\n', file=sys.stdout):
```

Petit exercice : faire une fonction `afficher` identique à `print`, c'est-à-dire prenant un nombre indéterminé de paramètres, les affichant en les séparant à l'aide du paramètre nommé `sep` et terminant l'affichage par la variable `fin`. Notre fonction `afficher` ne comptera pas de paramètre `file`. En outre, elle devra passer par `print` pour afficher (on ne connaît pas encore d'autres façons de faire). La seule contrainte est que l'appel à `print` ne doit compter qu'un seul paramètre non nommé. Autrement dit, avant l'appel à `print`, la chaîne devra avoir été déjà formatée, prête à l'affichage.

Pour que ce soit plus clair, je vous mets la définition de la fonction, ainsi que la `docstring` que j'ai écrite :

```
def afficher(*parametres, sep=' ', fin='\n'):
    """Fonction chargée de reproduire le comportement de print.

    Elle doit finir par faire appel à print pour afficher le ré
        sultat.
    Mais les paramètres devront déjà avoir été formatés.
    On doit passer à print une unique chaîne, en lui spécifiant
         de ne rien mettre à la fin :

    print(chaine, end='')"""

    # Les paramètres sont sous la forme d'un tuple
    # Or on a besoin de les convertir
    # Mais on ne peut pas modifier un tuple
    # On a plusieurs possibilités, ici je choisis de convertir
        le tuple en liste
    parametres = list(parametres)
    # On va commencer par convertir toutes les valeurs en chaî
        ne
    # Sinon on va avoir quelques problèmes lors du join
    for i, parametre in enumerate(parametres):
        parametres[i] = str(parametre)
    # La liste des paramètres ne contient plus que des chaînes
        de caractères
    # À présent on va constituer la chaîne finale
    chaine = sep.join(parametres)
    # On ajoute le paramètre fin à la fin de la chaîne
    chaine += fin
    # On affiche l'ensemble
    print(chaine, end='')
```

J'espère que ce n'était pas trop difficile et que, si vous avez fait des erreurs, vous avez pu les comprendre.

Ce n'est pas du tout grave si vous avez réussi à coder cette fonction d'une manière différente. *En programmation, il n'y a pas qu'une solution, il y a des solutions.*

Transformer une liste en paramètres de fonction

C'est peut-être un peu moins fréquent mais vous devez connaître ce mécanisme puisqu'il complète parfaitement le premier. Si vous avez un tuple ou une liste contenant des paramètres qui doivent être passés à une fonction, vous pouvez très simplement les transformer en paramètres lors de l'appel. Le seul problème c'est que, côté démonstration, je me vois un peu limité.

```
>>> liste_des_parametres = [1, 4, 9, 16, 25, 36]
>>> print(*liste_des_parametres)
1 4 9 16 25 36
>>>
```

Ce n'est pas bien spectaculaire et pourtant c'est une fonctionnalité très puissante du langage. Là, on a une liste contenant des paramètres et on la transforme en une liste de paramètres de la fonction `print`. Donc, au lieu d'afficher la liste proprement dite, on affiche tous les nombres, séparés par des espaces. C'est exactement comme si vous aviez fait `print(1, 4, 9, 16, 25, 36)`.



Mais quel intérêt ? Cela ne change pas grand-chose et il est rare que l'on capture les paramètres d'une fonction dans une liste, non ?

Oui je vous l'accorde. Ici l'intérêt ne saute pas aux yeux. Mais un peu plus tard, vous pourrez tomber sur des applications où les fonctions sont utilisées sans savoir quels paramètres elles attendent réellement. Si on ne connaît pas la fonction que l'on appelle, c'est très pratique. Là encore, vous découvrirez cela dans les chapitres suivants ou dans certains projets. Essayez de garder à l'esprit ce mécanisme de transformation.

On utilise une étoile `*` dans les deux cas. Si c'est dans une définition de fonction, cela signifie que les paramètres fournis non attendus lors de l'appel seront capturés dans la variable, sous la forme d'un tuple. Si c'est dans un appel de fonction, au contraire, cela signifie que la variable sera décomposée en plusieurs paramètres envoyés à la fonction.

J'espère que vous êtes encore en forme, on attaque le point que je considère comme le plus dur de ce chapitre, mais aussi le plus intéressant. Gardez les yeux ouverts !

Les compréhensions de liste

Les compréhensions de liste (« *list comprehensions* » en anglais) sont un moyen de filtrer ou modifier une liste très simplement. La syntaxe est déconcertante au début mais vous allez voir que c'est très puissant.

Parcours simple

Les **compréhensions de liste** permettent de parcourir une liste en en renvoyant une seconde, modifiée ou filtrée. Pour l'instant, nous allons voir une simple modification.

```
1 >>> liste_origine = [0, 1, 2, 3, 4, 5]
2 >>> [nb * nb for nb in liste_origine]
3 [0, 1, 4, 9, 16, 25]
4 >>>
```

Étudions un peu la ligne 2 de ce code. Comme vous avez pu le deviner, elle signifie en langage plus conventionnel « Mettre au carré tous les nombres contenus dans la liste d'origine ». Nous trouvons dans l'ordre, entre les crochets qui sont les délimiteurs d'une instruction de compréhension de liste :

- `nb * nb` : la valeur de retour. Pour l'instant, on ne sait pas ce qu'est la variable `nb`, on sait juste qu'il faut la mettre au carré. Notez qu'on aurait pu écrire `nb**2`, cela revient au même.
- `for nb in liste_origine` : voilà d'où vient notre variable `nb`. On reconnaît la syntaxe d'une boucle `for`, sauf qu'on n'est pas habitué à la voir sous cette forme.

Quand Python interprète cette ligne, il va parcourir la liste d'origine et mettre chaque élément de la liste au carré. Il renvoie ensuite le résultat obtenu, sous la forme d'une liste qui est de la même longueur que celle d'origine. On peut naturellement capturer cette nouvelle liste dans une variable.

Filtrage avec un branchement conditionnel

On peut aussi filtrer une liste de cette façon :

```
>>> liste_origine = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [nb for nb in liste_origine if nb%2==0]
[2, 4, 6, 8, 10]
>>>
```

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfère que les valeurs paires. Au final, on se retrouve donc avec une liste deux fois plus petite que celle d'origine.

Mélangeons un peu tout cela

Il est possible de filtrer et modifier une liste assez simplement. Par exemple, on a une liste contenant les quantités de fruits stockées pour un magasin (je ne suis pas sectaire, vous pouvez prendre des hamburgers si vous préférez). Chaque semaine, le magasin va prendre dans le stock une certaine quantité de chaque fruit, pour la mettre en vente. À ce moment, le stock de chaque fruit diminue naturellement. Inutile, en conséquence, de garder les fruits qu'on n'a plus en stock.

Je vais un peu reformuler. On va avoir une liste simple, qui contiendra des entiers, précisant la quantité de chaque fruit (c'est abstrait, les fruits ne sont pas précisés). On va faire une compréhension de liste pour diminuer d'une quantité donnée toutes les valeurs de cette liste, et on en profite pour retirer celles qui sont inférieures ou égales à 0.

```
>>> qtt_a_retirer = 7 # On retire chaque semaine 7 fruits de chaque sorte
>>> fruits_stockes = [15, 3, 18, 21] # Par exemple 15 pommes, 3 melons...
>>> [nb_fruits-qtt_a_retirer for nb_fruits in fruits_stockes if nb_fruits>qtt_a_retirer]
[8, 11, 14]
>>>
```

Comme vous le voyez, le fruit de quantité 3 n'a pas survécu à cette semaine d'achats. Bien sûr, cet exemple n'est pas complet : on n'a aucun moyen fiable d'associer les nombres restants aux fruits. Mais vous avez un exemple de filtrage et modification d'une liste.

Prenez bien le temps de regarder ces exemples : au début, la syntaxe des compréhensions de liste n'est pas forcément simple. Faites des essais, c'est aussi le meilleur moyen de comprendre.

Nouvelle application concrète

De nouveau, c'est à vous de travailler.

Nous allons en gros reprendre l'exemple précédent, en le modifiant un peu pour qu'il soit plus cohérent. Nous travaillons toujours avec des fruits sauf que, cette fois, nous allons associer un nom de fruit à la quantité restant en magasin. Nous verrons au prochain chapitre comment le faire avec des dictionnaires ; pour l'instant on va se contenter de listes :

```
>>> inventaire = [
...     ("pommes", 22),
...     ("melons", 4),
...     ("poires", 18),
...     ("fraises", 76),
...     ("prunes", 51),
... ]
>>>
```

Recopiez cette liste. Elle contient des tuples, contenant chacun un couple : le nom du fruit et sa quantité en magasin.

Votre mission est de trier cette liste en fonction de la quantité de chaque fruit. Autrement dit, on doit obtenir quelque chose de similaire à :

```
[
```

```
    ("fraises", 76),
    ("prunes", 51),
    ("pommes", 22),
    ("poires", 18),
    ("melons", 4),
]
```

Pour ceux qui n'ont pas eu la curiosité de regarder dans la documentation des listes, je signale à votre attention la méthode `sort` qui permet de trier une liste. Vous pouvez également utiliser la fonction `sorted` qui prend en paramètre la liste à trier (ce n'est pas une méthode de liste, faites attention). `sorted` renvoie la liste triée sans modifier la liste d'origine, ce qui peut être utile dans certaines circonstances, précisément celle-ci. À vous de voir, vous pouvez y arriver par les deux méthodes.

Bien entendu, essayez de faire cet exercice en utilisant les compréhensions de liste.

Je vous donne juste un petit indice : vous ne pouvez trier la liste comme cela, il faut l'inverser (autrement dit, placer la quantité avant le nom du fruit) pour pouvoir ensuite la trier par quantité.

Voici la correction que je vous propose :

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in
    inventaire]
# On n'a plus qu'à trier dans l'ordre décroissant l'inventaire
    inversé
# On reconstitue l'inventaire trié
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in sorted(
    inventaire_inverse, \
     reverse=True)]
```

Cela marche et le traitement a été fait en deux lignes.

Vous pouvez trier l'inventaire inversé avant la reconstitution, si vous trouvez cela plus compréhensible. Il faut privilégier la lisibilité du code.

```
# On change le sens de l'inventaire, la quantité avant le nom
inventaire_inverse = [(qtt, nom_fruit) for nom_fruit,qtt in
    inventaire]
# On trie l'inventaire inversé dans l'ordre décroissant
inventaire_inverse.sort(reverse=True)
# Et on reconstitue l'inventaire
inventaire = [(nom_fruit, qtt) for qtt,nom_fruit in
    inventaire_inverse)]
```

Faites des essais, entraînez-vous, vous en aurez sans doute besoin, la syntaxe n'est pas très simple au début. Et évitez de tomber dans l'extrême aussi : certaines opérations ne sont pas faisables avec les compréhensions de listes ou alors elles sont trop condensées pour être facilement compréhensibles. Dans l'exemple précédent, on aurait très bien pu remplacer nos deux à trois lignes d'instructions par une seule, mais cela aurait été dur à lire. Ne sacrifiez pas la lisibilité pour le simple plaisir de raccourcir votre code.

En résumé

- On peut découper une chaîne en fonction d'un séparateur en utilisant la méthode `split` de la chaîne.
- On peut joindre une liste contenant des chaînes de caractères en utilisant la méthode de chaîne `join`. Cette méthode doit être appelée sur le séparateur.
- On peut créer des fonctions attendant un nombre inconnu de paramètres grâce à la syntaxe `def fonction_inconnue(*parametres):` (les paramètres passés se retrouvent dans le tuple *parametres*).
- Les *compréhensions de listes* permettent de parcourir et filtrer une séquence en en renvoyant une nouvelle.
- La syntaxe pour effectuer un filtrage est la suivante : `nouvelle_squence = [element for element in ancienne_squence if condition]`.

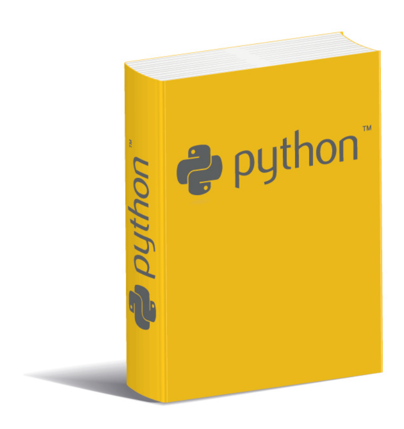
Chapitre 13

Les dictionnaires

Difficulté : 

Maintenant que vous commencez à vous familiariser avec la programmation orientée objet, nous allons pouvoir aller un peu plus vite sur les manipulations « classiques » de ce type, pour nous concentrer sur quelques petites spécificités propres aux dictionnaires.

Les dictionnaires sont des objets pouvant en contenir d'autres, à l'instar des listes. Cependant, au lieu d'héberger des informations dans un ordre précis, ils associent chaque objet contenu à une clé (la plupart du temps, une chaîne de caractères). Par exemple, un dictionnaire peut contenir un carnet d'adresses et on accède à chaque contact en précisant son nom.



Création et édition de dictionnaires

Un dictionnaire est un type de données extrêmement puissant et pratique. Il se rapproche des listes sur certains points mais, sur beaucoup d'autres, il en diffère totalement. Python utilise ce type pour représenter diverses fonctionnalités : on peut par exemple retrouver les attributs d'un objet grâce à un dictionnaire particulier.

Mais n'anticipons pas. Dans les deux chapitres précédents, nous avons découvert les listes. Les objets de ce type sont des objets conteneurs, dans lesquels on trouve d'autres objets. Pour accéder à ces objets contenus, il faut connaître leur position dans la liste. Cette position se traduit par des entiers, appelés indices, compris entre 0 (inclus) et la taille de la liste (non incluse). Tout cela, vous devez déjà le savoir.

Le dictionnaire est aussi un objet conteneur. Il n'a quant à lui aucune structure ordonnée, à la différence des listes. De plus, pour accéder aux objets contenus dans le dictionnaire, on n'utilise pas nécessairement des indices mais des **clés** qui peuvent être de bien des types distincts.

Créer un dictionnaire

Là encore, je vous donne le nom de la classe sur laquelle se construit un dictionnaire : `dict`. Vous devriez du même coup trouver la première méthode d'instanciation d'un dictionnaire :

```
>>> mon_dictionnaire = dict()
>>> type(mon_dictionnaire)
<class 'dict'>
>>> mon_dictionnaire
{}
>>> # Du coup, vous devriez trouver la deuxième manière de cré
    er un dictionnaire vide
... mon_dictionnaire = {}
>>> mon_dictionnaire
{}
>>>
```

Les parenthèses délimitent les tuples, les crochets délimitent les listes et les accolades `{}` délimitent les dictionnaires.

Voyons comment ajouter des clés et valeurs dans notre dictionnaire vide :

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire["pseudo"] = "Prolixe"
>>> mon_dictionnaire["mot de passe"] = "*"
>>> mon_dictionnaire
{'mot de passe': '*', 'pseudo': 'Prolixe'}
>>>
```

Nous indiquons entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée après le signe =. Sinon, l'ancienne valeur à l'emplacement indiqué est remplacée par la nouvelle :

```
1 >>> mon_dictionnaire = {}
2 >>> mon_dictionnaire["pseudo"] = "Prolixe"
3 >>> mon_dictionnaire["mot de passe"] = "*"
4 >>> mon_dictionnaire["pseudo"] = "6pri1"
5 >>> mon_dictionnaire
6 {'mot de passe': '*', 'pseudo': '6pri1'}
7 >>>
```

La valeur `'Prolixe'` pointée par la clé `'pseudo'` a été remplacée, à la ligne 4, par la valeur `'6pri1'`. Cela devrait vous rappeler la création de variables : si la variable n'existe pas, elle est créée, sinon elle est remplacée par la nouvelle valeur.

Pour accéder à la valeur d'une clé précise, c'est très simple :

```
>>> mon_dictionnaire["mot de passe"]
'*'
>>>
```

Si la clé n'existe pas dans le dictionnaire, une exception de type `KeyError` sera levée.

Généralisons un peu tout cela : nous avons des dictionnaires, qui peuvent contenir d'autres objets. On place ces objets et on y accède grâce à des clés. Un dictionnaire ne peut naturellement pas contenir deux clés identiques (comme on l'a vu, la seconde valeur écrase la première). En revanche, rien n'empêche d'avoir deux valeurs identiques dans le dictionnaire.

Nous avons utilisé ici, pour nos clés et nos valeurs, des chaînes de caractères. Ce n'est absolument pas obligatoire. Comme avec les listes, vous pouvez utiliser des entiers comme clés :

```
>>> mon_dictionnaire = {}
>>> mon_dictionnaire[0] = "a"
>>> mon_dictionnaire[1] = "e"
>>> mon_dictionnaire[2] = "i"
>>> mon_dictionnaire[3] = "o"
>>> mon_dictionnaire[4] = "u"
>>> mon_dictionnaire[5] = "y"
>>> mon_dictionnaire
{0: 'a', 1: 'e', 2: 'i', 3: 'o', 4: 'u', 5: 'y'}
>>>
```

On a l'impression de recréer le fonctionnement d'une liste mais ce n'est pas le cas : rappelez-vous qu'un dictionnaire n'a pas de structure ordonnée. Si vous supprimez par exemple l'indice 2, le dictionnaire, contrairement aux listes, ne va pas décaler toutes les clés d'indice supérieur à l'indice supprimé. Il n'a pas été conçu pour.

On peut utiliser quasiment tous les types comme clés et on peut utiliser absolument tous les types comme valeurs.

Voici un exemple un peu plus atypique de clés : on souhaite représenter un plateau d'échecs. Traditionnellement, on représente une case de l'échiquier par une lettre (de A à H) suivie d'un chiffre (de 1 à 8). La lettre définit la colonne et le chiffre définit la ligne. Si vous n'êtes pas sûrs de comprendre, regardez la figure 13.1.



FIGURE 13.1 – Échiquier

Pourquoi ne pas faire un dictionnaire dont les clés seront des tuples contenant la lettre et le chiffre identifiant la case, auxquelles on associe comme valeurs le nom des pièces ?

```
echiquier = {}
echiquier['a', 1] = "tour blanche" # En bas à gauche de l'échiquier
echiquier['b', 1] = "cavalier blanc" # À droite de la tour
echiquier['c', 1] = "fou blanc" # À droite du cavalier
echiquier['d', 1] = "reine blanche" # À droite du fou
# ... Première ligne des blancs
echiquier['a', 2] = "pion blanc" # Devant la tour
echiquier['b', 2] = "pion blanc" # Devant le cavalier, à droite du pion
# ... Seconde ligne des blancs
```

Dans cet exemple, nos tuples sont sous-entendus. On ne les place pas entre parenthèses. Python comprend qu'on veut créer des tuples, ce qui est bien, mais l'important est que vous le compreniez bien aussi. Certains cours encouragent à toujours placer des parenthèses autour des tuples quand on les utilise. Pour ma part, je pense que, si vous gardez à l'esprit qu'il s'agit de tuples, que vous n'avez aucune peine à l'identifier, cela

suffit. Si vous faites la confusion, mettez des parenthèses autour des tuples en toutes circonstances.

On peut aussi créer des dictionnaires déjà remplis :

```
placard = {"chemise":3, "pantalon":6, "tee-shirt":7}
```

On précise entre accolades la clé, le signe deux points « : » et la valeur correspondante. On sépare les différents couples `clé : valeur` par une virgule. C'est d'ailleurs comme cela que Python vous affiche un dictionnaire quand vous le lui demandez.

Certains ont peut-être essayé de créer des dictionnaires déjà remplis avant que je ne montre comment faire. Une petite précision, si vous avez tapé une instruction similaire à :

```
mon_dictionnaire = {'pseudo', 'mot de passe'}
```

Avec une telle instruction, ce n'est pas un dictionnaire que vous créez, mais un `set`.

Un `set` (ensemble) est un objet conteneur (lui aussi), très semblable aux listes sauf qu'il ne peut contenir deux objets identiques. Vous ne pouvez pas trouver deux fois dans un `set` l'entier `3` par exemple. Je vous laisse vous renseigner sur les `sets` si vous le désirez.

Supprimer des clés d'un dictionnaire

Comme pour les listes, vous avez deux possibilités mais elles reviennent sensiblement au même :

- le mot-clé `del` ;
- la méthode de dictionnaire `pop`.

Je ne vais pas m'attarder sur le mot-clé `del`, il fonctionne de la même façon que pour les listes :

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}
del placard["chemise"]
```

La méthode `pop` supprime également la clé précisée mais elle renvoie la valeur supprimée :

```
>>> placard = {"chemise":3, "pantalon":6, "tee shirt":7}
>>> placard.pop("chemise")
3
>>>
```

La méthode `pop` renvoie la valeur qui a été supprimée en même temps que la clé. Cela peut être utile parfois.

Voilà pour le tour d'horizon. Ce fut bref et vous n'avez pas vu toutes les méthodes, bien entendu. Je vous laisse consulter l'aide pour une liste détaillée.

Un peu plus loin

On se sert parfois des dictionnaires pour stocker des fonctions.

Je vais juste vous montrer rapidement le mécanisme sans trop m'y attarder. Là, je compte sur vous pour faire des tests si vous êtes intéressés. C'est encore un petit quelque chose que vous n'utiliserez peut-être pas tous les jours mais qu'il peut être utile de connaître.

Les fonctions sont manipulables comme des variables. Ce sont des objets, un peu particuliers mais des objets tout de même. Donc on peut les prendre pour valeur d'affectation ou les ranger dans des listes ou dictionnaires. C'est pourquoi je présente cette fonctionnalité à présent, auparavant j'aurais manqué d'exemples pratiques.

```
>>> print_2 = print # L'objet print_2 pointera sur la fonction
    print
>>> print_2("Affichons un message")
Affichons un message
>>>
```

On copie la fonction `print` dans une autre variable `print_2`. On peut ensuite appeler `print_2` et la fonction va afficher le texte saisi, tout comme `print` l'aurait fait.

En pratique, on affecte rarement des fonctions de cette manière. C'est peu utile. Par contre, on met parfois des fonctions dans des dictionnaires :

```
>>> def fete():
...     print("C'est la fête.")
...
>>> def oiseau():
...     print("Fais comme l'oiseau...")
...
>>> fonctions = {}
>>> fonctions["fete"] = fete # on ne met pas les parenthèses
>>> fonctions["oiseau"] = oiseau
>>> fonctions["oiseau"]
<function oiseau at 0x00BA5198>
>>> fonctions["oiseau"]() # on essaye de l'appeler
Fais comme l'oiseau...
>>>
```

Prenons dans l'ordre si vous le voulez bien :

- On commence par définir deux fonctions, `fete` et `oiseau` (pardonnez l'exemple).
- On crée un dictionnaire nommé `fonctions`.
- On met dans ce dictionnaire les fonctions `fete` et `oiseau`. La clé pointant vers la fonction est le nom de la fonction, tout bêtement, mais on aurait pu lui donner un nom plus original.
- On essaye d'accéder à la fonction `oiseau` en tapant `fonctions[« oiseau »]`. Python nous renvoie un truc assez moche, `<function oiseau at 0x00BA5198>`, mais vous

comprenez l'idée : c'est bel et bien notre fonction `oiseau`. Toutefois, pour l'appeler, il faut des parenthèses, comme pour toute fonction qui se respecte.
– En tapant `fonctions["oiseau"]()`, on accède à la fonction `oiseau` et on l'appelle dans la foulée.

On peut stocker les références des fonctions dans n'importe quel objet conteneur, des listes, des dictionnaires... et d'autres classes, quand nous apprendrons à en faire. Je ne vous demande pas de comprendre absolument la manipulation des références des fonctions, essayez simplement de retenir cet exemple. Dans tous les cas, nous aurons l'occasion d'y revenir.

Les méthodes de parcours

Comme vous pouvez le penser, le parcours d'un dictionnaire ne s'effectue pas tout à fait comme celui d'une liste. La différence n'est pas si énorme que cela mais, la plupart du temps, on passe par des méthodes de dictionnaire.

Parcours des clés

Peut-être avez-vous déjà essayé par vous-mêmes de parcourir un dictionnaire comme on l'a fait pour les listes :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for cle in fruits:
...     print(cle)
...
melons
poires
pommes
>>>
```

Comme vous le voyez, si on essaye de parcourir un dictionnaire « simplement », on parcourt en réalité la liste des clés contenues dans le dictionnaire.



Mais... les clés ne s'affichent pas dans l'ordre dans lequel on les a entrées... c'est normal ?

Les dictionnaires n'ont pas de structure ordonnée, gardez-le à l'esprit. Donc en ce sens oui, c'est tout à fait normal.

Une méthode de la classe `dict` permet d'obtenir ce même résultat. Personnellement, je l'utilise plus fréquemment car on est sûr, en lisant l'instruction, que c'est la liste des clés que l'on parcourt :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
```

```
>>> for cle in fruits.keys():
...     print(cle)
...
melons
poires
pommes
>>>
```

La méthode `keys` (« clés » en anglais) renvoie la liste des clés contenues dans le dictionnaire. En vérité, ce n'est pas tout à fait une liste (essayez de taper `fruits.keys()` dans votre interpréteur) mais c'est une séquence qui se parcourt comme une liste.

Parcours des valeurs

On peut aussi parcourir les valeurs contenues dans un dictionnaire. Pour ce faire, on utilise la méthode `values` (« valeurs » en anglais).

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for valeur in fruits.values():
...     print(valeur)
...
3
31
21
>>>
```

Cette méthode est peu utilisée pour un parcours car il est plus pratique de parcourir la liste des clés, cela suffit pour avoir les valeurs correspondantes. Mais on peut aussi, bien entendu, l'utiliser dans une condition :

```
>>> if 21 in fruits.values():
...     print("Un des fruits se trouve dans la quantité 21.")
...
Un des fruits se trouve dans la quantité 21.
>>>
```

Parcours des clés et valeurs simultanément

Pour avoir en même temps les indices et les objets d'une liste, on utilise la fonction `enumerate`, j'espère que vous vous en souvenez. Pour faire de même avec les dictionnaires, on utilise la méthode `items`. Elle renvoie une liste, contenant les couples `clé : valeur`, sous la forme d'un `tuple`. Voyons comment l'utiliser :

```
>>> fruits = {"pommes":21, "melons":3, "poires":31}
>>> for cle, valeur in fruits.items():
...     print("La clé {} contient la valeur {}.".format(cle,
    valeur))
```

```
...
La clé melons contient la valeur 3.
La clé poires contient la valeur 31.
La clé pommes contient la valeur 21.
>>>
```

Il est parfois très pratique de parcourir un dictionnaire avec ses clés et les valeurs associées.

Entraînez-vous, il n'y a que cela de vrai. Pourquoi pas reprendre l'exercice du chapitre précédent, avec notre inventaire de fruits ? Sauf que le type de l'inventaire ne serait pas une liste mais un dictionnaire associant les noms des fruits aux quantités ?

Il nous reste une petite fonctionnalité supplémentaire à voir et on en aura fini avec les dictionnaires.

Les dictionnaires et paramètres de fonction

Cela ne vous rappelle pas quelque chose ? J'espère bien que si, on a vu quelque chose de similaire au chapitre précédent.

Si vous vous souvenez, on avait réussi à intercepter tous les paramètres de la fonction… sauf les paramètres nommés.

Récupérer les paramètres nommés dans un dictionnaire

Il existe aussi une façon de capturer les paramètres nommés d'une fonction. Dans ce cas, toutefois, ils sont placés dans un dictionnaire. Si, par exemple, vous appelez la fonction ainsi : `fonction(parametre='a')`, vous aurez, dans le dictionnaire capturant les paramètres nommés, une clé `'parametre'` liée à la valeur `'a'`. Voyez plutôt :

```
>>> def fonction_inconnue(**parametres_nommes):
...     """Fonction permettant de voir comment récupérer les paramètres nommés
...     dans un dictionnaire"""
...
...
...     print("J'ai reçu en paramètres nommés : {}.".format(parametres_nommes))
...
>>> fonction_inconnue() # Aucun paramètre
J'ai reçu en paramètres nommés : {}
>>> fonction_inconnue(p=4, j=8)
J'ai reçu en paramètres nommés : {'p': 4, 'j': 8}
>>>
```

Pour capturer tous les paramètres nommés non précisés dans un dictionnaire, il faut mettre deux étoiles `**` avant le nom du paramètre.

Si vous passez des paramètres non nommés à cette fonction, Python lèvera une exception.

Ainsi, pour avoir une fonction qui accepte n'importe quel type de paramètres, nommés ou non, dans n'importe quel ordre, dans n'importe quelle quantité, il faut la déclarer de cette manière :

```
def fonction_inconnue(*en_liste, **en_dictionnaire):
```

Tous les paramètres non nommés se retrouveront dans la variable `en_liste` et les paramètres nommés dans la variable `en_dictionnaire`.



Mais à quoi cela peut-il bien servir d'avoir une fonction qui accepte n'importe quel paramètre ?

Pour l'instant à pas grand chose mais cela viendra. Quand on abordera le chapitre sur les décorateurs, vous vous en souviendrez et vous pourrez vous féliciter de connaître cette fonctionnalité.

Transformer un dictionnaire en paramètres nommés d'une fonction

Là encore, on peut faire exactement l'inverse : transformer un dictionnaire en paramètres nommés d'une fonction. Voyons un exemple tout simple :

```
>>> parametres = {"sep":" >> ", "end":" -\n"}
>>> print("Voici", "un", "exemple", "d'appel", **parametres)
Voici >> un >> exemple >> d'appel -
>>>
```

Les paramètres nommés sont transmis à la fonction par un dictionnaire. Pour indiquer à Python que le dictionnaire doit être transmis comme des paramètres nommés, on place deux étoiles avant son nom `**` dans l'appel de la fonction.

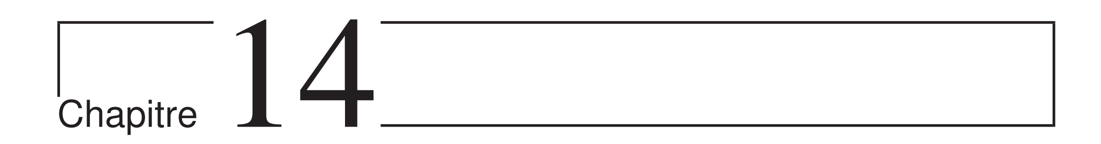
Comme vous pouvez le voir, c'est comme si nous avions écrit :

```
>>> print("Voici", "un", "exemple", "d'appel", sep=" >> ", end=
    " -\n")
Voici >> un >> exemple >> d'appel -
>>>
```

Pour l'instant, vous devez trouver que c'est bien se compliquer la vie pour si peu. Nous verrons dans la suite de ce cours qu'il n'en est rien, en fait, même si nous n'utilisons pas cette fonctionnalité tous les jours.

En résumé

- Un dictionnaire est un objet conteneur associant des clés à des valeurs.
- Pour créer un dictionnaire, on utilise la syntaxe `dictionnaire = {cle1:valeur1, cle2=valeur2, cleN=valeurN}`.
- On peut ajouter ou remplacer un élément dans un dictionnaire : `dictionnaire[cle] = valeur`.
- On peut supprimer une clé (et sa valeur correspondante) d'un dictionnaire en utilisant, au choix, le mot-clé `del` ou la méthode `pop`.
- On peut parcourir un dictionnaire grâce aux méthodes `keys` (parcourt les clés), `values` (parcourt les valeurs) ou `items` (parcourt les couples clé-valeur).
- On peut capturer les paramètres nommés passés à une fonction en utilisant cette syntaxe : `def fonction_inconnue(**parametres_nommes` : (les paramètres nommés se retrouvent dans le dictionnaire `parametres_nommes`).



Chapitre 14

Les fichiers

Difficulté :

Poursuivons notre tour d'horizon des principaux objets. Nous allons voir dans ce chapitre les fichiers, comment les ouvrir, les lire, écrire dedans.

Nous finirons ce chapitre en voyant comment sauvegarder nos objets dans des fichiers, afin de les utiliser d'une session à l'autre de notre programme.



Avant de commencer

Nous allons beaucoup travailler sur des répertoires et des fichiers, autrement dit sur votre disque. Donc je vais vous donner quelques informations générales avant de commencer pour que, malgré vos différents systèmes et configurations, vous puissiez essayer les instructions que je vais vous montrer.

Mais d'abord, pourquoi lire ou écrire dans des fichiers ?

Peut-être que vous ne voyez pas trop l'intérêt de savoir lire et écrire dans des fichiers, hormis quelques applications de temps à autre. Mais souvenez-vous que, quand vous fermez votre programme, aucune de vos variables n'est sauvegardée. Or, les fichiers peuvent être, justement, un excellent moyen de garder les valeurs de certains objets pour pouvoir les récupérer quand vous rouvrirez votre programme. Par exemple, un petit jeu peut enregistrer les scores des joueurs.

Si, dans notre TP **ZCasino**, nous avions pu enregistrer la somme que nous avions en poche au moment de quitter le casino, nous aurions pu rejouer sans repartir de zéro.

Changer le répertoire de travail courant

Si vous souhaitez travailler dans l'interpréteur Python, et je vous y encourage, vous devrez changer le répertoire de travail courant. En effet, au lancement de l'interpréteur, le répertoire de travail courant est celui dans lequel se trouve l'exécutable de l'interpréteur. Sous Windows, c'est `C:\Python3X`, le X étant différent en fonction de votre version de Python. Dans tous les cas, je vous invite à changer de répertoire de travail courant. Pour cela, vous devez utiliser une fonction du module `os`, qui s'appelle `chdir` `(Change Directory)`.

```
>>> import os
>>> os.chdir("C:/tests python")
>>>
```

Pour que cette instruction fonctionne, le répertoire doit exister. Modifiez la chaîne passée en paramètre de `os.chdir` en fonction du dossier dans lequel vous souhaitez vous déplacer.



Je vous conseille, que vous soyez sous Windows ou non, d'utiliser le symbole / pour décrire un chemin.

Vous pouvez utiliser, en le doublant, l'antislash \\ mais, si vous oubliez de le doubler, vous aurez des erreurs. Je vous conseille donc d'utiliser le slash /, cela fonctionne très bien même sous Windows.



Quand vous lancez un programme Python directement, par exemple en faisant un double-clic dessus, le répertoire courant est celui d'où vous lancez le programme. Si vous avez un fichier `mon_programme.py` contenu sur le disque `C:`, le répertoire de travail courant quand vous lancerez le programme sera `C:\`.

Chemins relatifs et absolus

Pour décrire l'arborescence d'un système, on a deux possibilités :

- les chemins absolus ;
- les chemins relatifs.

Le chemin absolu

Quand on décrit une cible (un fichier ou un répertoire) sous la forme d'un chemin absolu, on décrit la suite des répertoires menant au fichier. Sous Windows, on partira du nom de volume (`C:\`, `D:\`...). Sous les systèmes Unix, ce sera plus vraisemblablement depuis /.

Par exemple, sous Windows, si on a un fichier nommé `fic.txt`, contenu dans un dossier `test`, lui-même présent sur le disque `C:`, le chemin absolu menant à notre fichier sera `C:\test\fic.txt`.

Le chemin relatif

Quand on décrit la position d'un fichier grâce à un chemin relatif, cela veut dire que l'on tient compte du dossier dans lequel on se trouve actuellement. Ainsi, si on se trouve dans le dossier `C:\test` et que l'on souhaite accéder au fichier `fic.txt` contenu dans ce même dossier, le chemin relatif menant à ce fichier sera tout simplement `fic.txt`.

Maintenant, si on se trouve dans `C:`, notre chemin relatif sera `test\fic.txt`.

Quand on décrit un chemin relatif, on utilise parfois le symbole `..` qui désigne le répertoire parent. Voici un nouvel exemple :

- `C:`
  - `test`
    - `rep1`
      - `fic1.txt`
    - `rep2`
      - `fic2.txt`
      - `fic3.txt`

C'est dans notre dossier `test` que tout se passe. Nous avons deux sous-répertoires nommés `rep1` et `rep2`. Dans `rep1`, nous avons un seul fichier : `fic1.txt`. Dans `rep2`, nous avons deux fichiers : `fic2.txt` et `fic3.txt`.

Si le répertoire de travail courant est `rep2` et que l'on souhaite accéder à `fic1.txt`, notre chemin relatif sera donc `..\rep1\fic1.txt`.



J'utilise ici des anti-slash parce que l'exemple d'arborescence est un modèle Windows et que ce sont les séparateurs utilisés pour décrire une arborescence Windows. Mais, dans votre code je vous conseille quand même d'utiliser un slash (/).

Résumé

Les chemins absolus et relatifs sont donc deux moyens de décrire le chemin menant à des fichiers ou répertoires. Mais, si le résultat est le même, le moyen utilisé n'est pas identique : quand on utilise un chemin absolu, on décrit l'intégralité du chemin menant au fichier, peu importe l'endroit où on se trouve. Un chemin absolu permet d'accéder à un endroit dans le disque quel que soit le répertoire de travail courant. L'inconvénient de cette méthode, c'est qu'on doit préalablement savoir où se trouvent, sur le disque, les fichiers dont on a besoin.

Le chemin relatif décrit la succession de répertoires à parcourir en prenant comme point d'origine non pas la racine, ou le périphérique sur lequel est stockée la cible, mais le répertoire dans lequel on se trouve. Cela présente certains avantages quand on code un projet, on n'est pas obligé de savoir où le projet est stocké pour construire plusieurs répertoires. Mais ce n'est pas forcément la meilleure solution en toutes circonstances.

Comme je l'ai dit, quand on lance l'interpréteur Python, on a bel et bien un répertoire de travail courant. Vous pouvez l'afficher grâce à la fonction `os.getcwd()` [1].

Cela devrait donc vous suffire. Pour les démonstrations qui vont suivre, placez-vous, à l'aide de `os.chdir`, dans un répertoire de test créé pour l'occasion.

Lecture et écriture dans un fichier

Nous allons commencer à lire avant d'écrire dans un fichier. Pour l'exemple donc, je vous invite à créer un fichier dans le répertoire de travail courant que vous avez choisi. Je suis en manque flagrant d'inspiration, je vais l'appeler `fichier.txt` et je vais écrire dedans, à l'aide d'un éditeur sans mise en forme (tel que le bloc-notes Windows) : *« C'est le contenu du fichier. Spectaculaire non ? »*

Ouverture du fichier

D'abord, il nous faut ouvrir le fichier avec Python. On utilise pour ce faire la fonction `open`, disponible sans avoir besoin de rien importer. Elle prend en paramètre :

– le chemin (absolu ou relatif) menant au fichier à ouvrir ;

1. CWD = « Current Working Directory »

– le mode d'ouverture.

Le mode est donné sous la forme d'une chaîne de caractères. Voici les principaux modes :

- `'r'` : ouverture en lecture (Read).
- `'w'` : ouverture en écriture (Write). Le contenu du fichier est écrasé. Si le fichier n'existe pas, il est créé.
- `'a'` : ouverture en écriture en mode ajout (Append). On écrit à la fin du fichier sans écraser l'ancien contenu du fichier. Si le fichier n'existe pas, il est créé.

On peut ajouter à tous ces modes le signe `b` pour ouvrir le fichier en mode binaire. Nous en verrons plus loin l'utilité, c'est un mode un peu particulier.

Ici nous souhaitons lire le fichier. Nous allons donc utiliser le mode `'r'`.

```
>>> mon_fichier = open("fichier.txt", "r")
>>> mon_fichier
<_io.TextIOWrapper name='fichier.txt' encoding='cp1252'>
>>> type(mon_fichier)
<class '_io.TextIOWrapper'>
>>>
```

L'encodage précisé quand on affiche le fichier dans l'interpréteur peut être très différent suivant votre système. Ici, je suis dans l'interpréteur Python dans Windows et l'encodage choisi est donc un encodage Windows propre à la console. Ne soyez pas surpris s'il est différent chez vous.

La fonction `open` crée donc un fichier. Elle renvoie un objet de la classe `TextIoWrapper`. Par la suite, nous allons utiliser des méthodes de cette classe pour interagir avec le fichier.

Le type de l'objet doit vous surprendre quelque peu. Cela aurait très bien pu être un type `file` après tout. En fait, `open` permet d'ouvrir un fichier, mais `TextIoWrapper` est utilisé dans d'autres circonstances, pour afficher du texte à l'écran par exemple. Bon, cela ne nous concerne pas trop ici, je ne vais pas m'y attarder.

Fermer le fichier

N'oubliez pas de fermer un fichier après l'avoir ouvert. Si d'autres applications, ou d'autres morceaux de votre propre code, souhaitent accéder à ce fichier, ils ne pourront pas car le fichier sera déjà ouvert. C'est surtout vrai en écriture, mais prenez de bonnes habitudes. La méthode à utiliser est `close` :

```
>>> mon_fichier.close()
>>>
```

Lire l'intégralité du fichier

Pour ce faire, on utilise la méthode `read` de la classe `TextIoWrapper`. Elle renvoie l'intégralité du fichier :

```
>>> mon_fichier = open("fichier.txt", "r")
>>> contenu = mon_fichier.read()
>>> print(contenu)
C'est le contenu du fichier. Spectaculaire non ?
>>> mon_fichier.close()
>>>
```

Quoi de plus simple ? La méthode `read` renvoie tout le contenu du fichier, que l'on capture dans une chaîne de caractères. Notre fichier ne contient pas de saut de ligne mais, si c'était le cas, vous auriez dans votre variable `contenu` les signes `\n` traduisant un saut de ligne.

Maintenant que vous avez une chaîne, vous pouvez naturellement tout faire : la convertir, tout entière ou en partie, si c'est nécessaire, `split` la chaîne pour parcourir chaque ligne et les traiter... bref, tout est possible.

Écriture dans un fichier

Bien entendu, il nous faut ouvrir le fichier avant tout. Vous pouvez utiliser le mode `w` ou le mode `a`. Le premier écrase le contenu éventuel du fichier, alors que le second ajoute ce que l'on écrit à la fin du fichier. À vous de voir en fonction de vos besoins. Dans tous les cas, ces deux modes créent le fichier s'il n'existe pas.

Écrire une chaîne

Pour écrire dans un fichier, on utilise la méthode `write` en lui passant en paramètre la chaîne à écrire dans le fichier. Elle renvoie le nombre de caractères qui ont été écrits. On n'est naturellement pas obligé de récupérer cette valeur, sauf si on en a besoin.

```
>>> mon_fichier = open("fichier.txt", "w") # Argh j'ai tout é
    crasé !
>>> mon_fichier.write("Premier test d'écriture dans un fichier
    via Python")
50
>>> mon_fichier.close()
>>>
```

Vous pouvez vérifier que votre fichier contient bien le texte qu'on y a écrit.

Écrire d'autres types de données

La méthode `write` n'accepte en paramètre que des chaînes de caractères. Si vous voulez écrire dans votre fichier des nombres, des scores par exemple, il vous faudra les convertir en chaîne avant de les écrire et les convertir en entier après les avoir lus.

Le module `os` contient beaucoup de fonctions intéressantes pour créer et supprimer des fichiers et des répertoires. Je vous laisse regarder l'aide si vous êtes intéressé.

Le mot-clé `with`

Ne désespérez pas, il ne nous reste plus autant de mots-clés à découvrir... mais quelques-uns tout de même. Et même certains dont je ne parlerai pas...

On n'est jamais à l'abri d'une erreur. Surtout quand on manipule des fichiers. Il peut se produire des erreurs quand on lit, quand on écrit... et si l'on n'y prend garde, le fichier restera ouvert.

Comme je vous l'ai dit, c'est plutôt gênant et cela peut même être grave. Si votre programme souhaite de nouveau utiliser ce fichier, il ne pourra pas forcément y accéder, puisqu'il a déjà été ouvert.

Il existe un mot-clé qui permet d'éviter cette situation : `with`. Voici sa syntaxe :

```
with open(mon_fichier, mode_ouverture) as variable:
# Opérations sur le fichier
```

On trouve dans l'ordre :

- Le mot-clé `with`, prélude au bloc dans lequel on va manipuler notre fichier. On peut trouver `with` dans la manipulation d'autres objets mais nous ne le verrons pas ici.
- Notre objet. Ici, on appelle `open` qui va renvoyer un objet `TextIOWrapper` (notre fichier).
- Le mot-clé `as` que nous avons déjà vu dans le mécanisme d'importation et dans les exceptions. Il signifie toujours la même chose : « en tant que ».
- Notre variable qui contiendra notre objet. Si la variable n'existe pas, Python la crée.

Un exemple ?

```
>>> with open('fichier.txt', 'r') as mon_fichier:
...     texte = mon_fichier.read()
...
>>>
```



Cela ne veut pas dire que le bloc d'instructions ne lèvera aucune exception.

Cela signifie simplement que, si une exception se produit, le fichier sera tout de même fermé à la fin du bloc.

Vous pouvez appeler `mon_fichier.closed` pour le vérifier. Si le fichier est fermé, `mon_fichier.closed` vaudra `True`.

Il est inutile, par conséquent, de fermer le fichier à la fin du bloc `with`. Python va le faire tout seul, qu'une exception soit levée ou non. Je vous encourage à utiliser cette syntaxe, elle est plus sûre et plus facile à comprendre.

Allez ! Direction le module `pickle`, dans lequel nous allons apprendre à sauvegarder nos objets dans des fichiers.

Enregistrer des objets dans des fichiers

Dans beaucoup de langages de haut niveau, on peut enregistrer ses objets dans un fichier. Python ne fait pas exception. Grâce au module `pickle` que nous allons découvrir, on peut enregistrer n'importe quel objet et le récupérer par la suite, au prochain lancement du programme, par exemple. En outre, le fichier résultant pourra être lu depuis n'importe quel système d'exploitation (à condition, naturellement, que celui-ci prenne en charge Python).

Enregistrer un objet dans un fichier

Il nous faut naturellement d'abord importer le module `pickle`.

```
>>> import pickle
>>>
```

On va ensuite utiliser deux classes incluses dans ce module : la classe `Pickler` et la classe `Unpickler`.

C'est la première qui nous intéresse dans cette section.

Pour créer notre objet `Pickler`, nous allons l'appeler en passant en paramètre le fichier dans lequel nous allons enregistrer notre objet.

```
>>> with open('donnees', 'wb') as fichier:
...     mon_pickler = pickle.Pickler(fichier)
...     # enregistrement ...
...
>>>
```

Quand nous allons enregistrer nos objets, ce sera dans le fichier `donnees`. Je ne lui ai pas donné d'extension, vous pouvez le faire. Mais évitez de préciser une extension qui est utilisée par un programme.

Notez le mode d'ouverture : on ouvre le fichier `donnees` en mode d'écriture binaire. Il suffit de rajouter, derrière la lettre symbolisant le mode, la lettre `b` pour indiquer un mode binaire.

Le fichier que Python va écrire ne sera pas très lisible si vous essayez de l'ouvrir, mais ce n'est pas le but.

Bon. Maintenant que notre pickler est créé, nous allons enregistrer un ou plusieurs objets dans notre fichier. Là, c'est à vous de voir comment vous voulez vous organiser, cela dépend aussi beaucoup du projet. Moi, j'ai pris l'habitude de n'enregistrer qu'un objet par fichier, mais il n'y a aucune obligation.

On utilise la méthode `dump` du pickler pour enregistrer l'objet. Son emploi est des plus simples :

```
>>> score = {
...     "joueur 1":    5,
...     "joueur 2":   35,
...     "joueur 3":   20,
...     "joueur 4":    2,
>>> }
>>> with open('donnees', 'wb') as fichier:
...     mon_pickler = pickle.Pickler(fichier)
...     mon_pickler.dump(score)
...
>>>
```

Après l'exécution de ce code, vous avez dans votre dossier de test un fichier `donnees` qui contient... eh bien, notre dictionnaire contenant les scores de nos quatre joueurs. Si vous voulez enregistrer plusieurs objets, appelez de nouveau la méthode `dump` avec les objets à enregistrer. Ils seront ajoutés dans le fichier dans l'ordre où vous les enregistrez.

Récupérer nos objets enregistrés

Nous allons utiliser une autre classe définie dans notre module `pickle`. Cette fois, assez logiquement, c'est la classe `Unpickler`.

Commençons par créer notre objet. À sa création, on lui passe le fichier dans lequel on va lire les objets. Puisqu'on va lire, on change de mode, on repasse en mode `r`, et même `rb` puisque le fichier est binaire.

```
>>> with open('donnees', 'rb') as fichier:
...     mon_depickler = pickle.Unpickler(fichier)
...     # Lecture des objets contenus dans le fichier...
...
>>>
```

Pour lire l'objet dans notre fichier, il faut appeler la méthode `load` de notre depickler. Elle renvoie le premier objet qui a été lu (s'il y en a plusieurs, il faut l'appeler plusieurs fois).

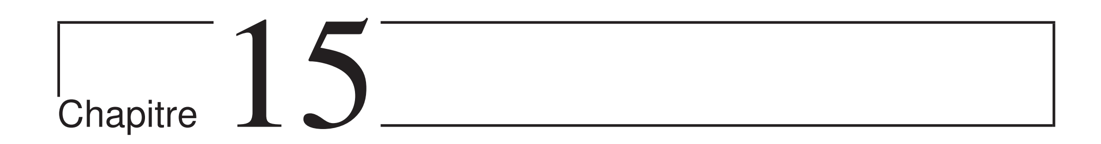
```
>>> with open('donnees', 'rb') as fichier:
...     mon_depickler = pickle.Unpickler(fichier)
```

```
...      score_recupere = mon_depickler.load()
...
>>>
```

Et après cet appel, si le fichier a pu être lu, dans votre variable `score_recupere`, vous récupérez votre dictionnaire contenant les scores. Là, c'est peut-être peu spectaculaire mais, quand vous utilisez ce module pour sauvegarder des objets devant être conservés alors que votre programme n'est pas lancé, c'est franchement très pratique.

En résumé

- On peut ouvrir un fichier en utilisant la fonction `open` prenant en paramètre le chemin vers le fichier et le mode d'ouverture.
- On peut lire dans un fichier en utilisant la méthode `read`.
- On peut écrire dans un fichier en utilisant la méthode `write`.
- Un fichier doit être refermé après usage en utilisant la méthode `close`.
- Le module `pickle` est utilisé pour enregistrer des objets Python dans des fichiers et les recharger ensuite.



Chapitre 15

Portée des variables et références

Difficulté : 

Dans ce chapitre, je vais m'attarder sur la portée des variables et sur les références. Je ne vais pas vous faire une visite guidée de la mémoire de votre ordinateur (Python est assez haut niveau pour, justement, ne pas avoir à descendre aussi bas), je vais simplement souligner quelques cas intéressants que vous pourriez rencontrer dans vos programmes.

Ce chapitre n'est pas indispensable, mais je ne l'écris naturellement pas pour le plaisir : vous pouvez très bien continuer à apprendre le Python sans connaître précisément comment Python joue avec les références, mais il peut être utile de le savoir.

N'hésitez pas à relire ce chapitre si vous avez un peu de mal, les concepts présentés ne sont pas évidents.



La portée des variables

En Python, comme dans la plupart des langages, on trouve des règles qui définissent la **portée des variables**. La portée utilisée dans ce sens c'est « quand et comment les variables sont-elles accessibles ? ». Quand vous définissez une fonction, quelles variables sont utilisables dans son corps ? Uniquement les paramètres ? Est-ce qu'on peut créer dans notre corps de fonction des variables utilisables en dehors ? Si vous ne vous êtes jamais posé ces questions, c'est normal. Mais je vais tout de même y répondre car elles ne sont pas dénuées d'intérêt.

Dans nos fonctions, quelles variables sont accessibles ?

On ne change pas une équipe qui gagne : passons aux exemples dès à présent.

```
>>> a = 5
>>> def print_a():
...     """Fonction chargée d'afficher la variable a.
...     Cette variable a n'est pas passée en paramètre de la fonction.
...     On suppose qu'elle a été créée en dehors de la fonction, on veut voir
...     si elle est accessible depuis le corps de la fonction"""
...
...     print("La variable a = {0}.".format(a))
...
>>> print_a()
La variable a = 5.
>>> a = 8
>>> print_a()
La variable a = 8.
>>>
```

Surprise ! Ou peut-être pas...

La variable `a` n'est pas passée en paramètre de la fonction `print_a`. Et pourtant, Python la trouve, tant qu'elle a été définie avant **l'appel** de la fonction.

C'est là qu'interviennent les différents espaces.

L'espace local

Dans votre fonction, quand vous faites référence à une variable `a`, Python vérifie dans l'**espace local** de la fonction. Cet espace contient les paramètres qui sont passés à la fonction et les variables définies dans son corps. Python apprend ainsi que la variable `a` n'existe pas dans l'espace local de la fonction. Dans ce cas, il va regarder dans l'espace local dans lequel la fonction a été appelée. Et là, il trouve bien la variable `a` et peut donc l'afficher.

D'une façon générale, je vous conseille d'éviter d'appeler des variables qui ne sont pas dans l'espace local, sauf si c'est nécessaire. Ce n'est pas très clair à la lecture ; dans l'absolu, préférez travailler sur des variables globales, cela reste plus propre (nous verrons cela plus bas). Pour l'instant, on ne s'intéresse qu'aux mécanismes, on cherche juste à savoir quelles variables sont accessibles depuis le corps d'une fonction et de quelle façon.

La portée de nos variables

Voyons quelques cas concrets. Je vais les expliquer au fur et à mesure, ne vous en faites pas.

Qu'advient-il des variables définies dans un corps de fonction ?

Voyons un nouvel exemple :

```
def set_var(nouvelle_valeur):
    """Fonction nous permettant de tester la portée des variables
    définies dans notre corps de fonction"""

    # On essaye d'afficher la variable var, si elle existe
    try:
        print("Avant l'affectation, notre variable var vaut {0}.".format(var))
    except NameError:
        print("La variable var n'existe pas encore.")
    var = nouvelle_valeur
    print("Après l'affectation, notre variable var vaut {0}.".format(var))
```

Et maintenant, utilisons notre fonction :

```
>>> set_var(5)
La variable var n'existe pas encore.
Après l'affectation, notre variable var vaut 5.
>>> var
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined
>>>
```

Je sens que quelques explications s'imposent :

– Lors de notre appel à `set_var`, notre variable `var` n'a pu être trouvée par Python : c'est normal, nous ne l'avons pas encore définie, ni dans notre corps de fonction, ni dans le corps de notre programme. Python affecte la valeur `5` à la variable `var`, l'affiche et s'arrête.

– Au sortir de la fonction, on essaye d'afficher la variable `var`... mais Python ne la trouve pas ! En effet : elle a été définie dans le corps de la fonction (donc dans son espace local) et, à la fin de l'exécution de la fonction, l'espace est détruit... donc la variable `var`, définie dans le corps de la fonction, n'existe que dans ce corps et est détruite ensuite.

Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier. C'est pourquoi, dans notre fonction `print_a`, on arrivait à afficher une variable qui n'était pas comprise dans l'espace local de la fonction. En revanche, on ne peut modifier la valeur d'une variable extérieure à l'espace local, par affectation du moins. Si dans votre corps de fonction vous faites `var = nouvelle_valeur`, vous n'allez *en aucun cas* modifier une variable extérieure au corps.

En fait, quand Python trouve une instruction d'affectation, comme par exemple `var = nouvelle_valeur`, il va changer la valeur de la variable dans l'espace local de la fonction. Et rappelez-vous que cet espace local est détruit après l'appel à la fonction.

Pour résumer, et c'est ce qu'il faut retenir, *une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.*

Cela paraît plutôt stupide au premier abord... mais pas d'impatience. Je vais relativiser cela assez rapidement.

Une fonction modifiant des objets

J'espère que vous vous en souvenez, *en Python, tout est objet.* Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis. Et pas les valeurs des objets, mais bien les objets eux-mêmes, ceci est très important.

Bon. On ne peut affecter une nouvelle valeur à un paramètre dans le corps de la fonction. Je ne reviens pas là-dessus. En revanche, on pourrait essayer d'appeler une méthode de l'objet qui le modifie... Voyons cela :

```
>>> def ajouter(liste, valeur_a_ajouter):
...     """Cette fonction insère à la fin de la liste la valeur
    que l'on veut ajouter"""
...     liste.append(valeur_a_ajouter)
...
>>> ma_liste=['a', 'e', 'i']
>>> ajouter(ma_liste, 'o')
>>> ma_liste
['a', 'e', 'i', 'o']
>>>
```

Cela marche ! On passe en paramètres notre objet de type `list` avec la valeur à ajouter. Et la fonction appelle la méthode `append` de l'objet. Cette fois, au sortir de la fonction, notre objet a bel et bien été modifié.



Je vois pas pourquoi. Tu as dit qu'une fonction ne pouvait pas affecter de nouvelles valeurs aux paramètres ?

Absolument. Mais c'est cela la petite subtilité dans l'histoire : on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet. Et cela change tout. Si vous vous embrouillez, retenez que, dans le corps de fonction, si vous faites `parametre = nouvelle_valeur`, le paramètre ne sera modifié que dans le corps de la fonction. Alors que si vous faites `parametre.methode_pour_modifier(...)`, l'objet derrière le paramètre sera bel et bien modifié.

On peut aussi modifier les attributs d'un objet, par exemple changer une case de la liste ou d'un dictionnaire : ces changements aussi seront effectifs au-delà de l'appel de la fonction.

Et les références, dans tout cela ?

J'ai parlé des références, et vous ai promis d'y consacrer une section ; c'est maintenant qu'on en parle !

Je vais schématiser volontairement : les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Concrètement, j'ai présenté les variables comme ceci : un nom identifiant pointant vers une valeur. Par exemple, notre variable nommée `a` possède une valeur (disons 0).

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Cela reste plus haut niveau que les pointeurs en C par exemple, ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.



Bah. . . bien sûr, rien n'empêche de faire deux variables avec la même valeur.

Non non, je ne parle pas de valeurs ici mais d'objets. Voyons un exemple, vous allez comprendre :

```
>>> ma_liste1 = [1, 2, 3]
>>> ma_liste2 = ma_liste1
>>> ma_liste2.append(4)
>>> print(ma_liste2)
[1, 2, 3, 4]
>>> print(ma_liste1)
[1, 2, 3, 4]
>>>
```

Nous créons une liste dans la variable `ma_liste1`. À la ligne 2, nous affectons `ma_liste1` à la variable `ma_liste2`. On pourrait croire que `ma_liste2` est une copie de `ma_liste1`. Toutefois, quand on ajoute 4 à `ma_liste2`, `ma_liste1` est aussi modifiée.

On dit que `ma_liste1` et `ma_liste2` contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.



Euh... j'essaye de faire la même chose avec des variables contenant des entiers et cela ne marche pas.

C'est normal. Les entiers, les flottants, les chaînes de caractères, n'ont aucune méthode travaillant sur l'objet lui-même. Les chaînes de caractères, comme nous l'avons vu, ne modifient pas l'objet appelant mais renvoient un nouvel objet modifié. Et comme nous venons de le voir, le processus d'affectation n'est pas du tout identique à un appel de méthode.



Et si je veux modifier une liste sans toucher à l'autre ?

Eh bien c'est impossible, vu comment nous avons défini nos listes. Les deux variables pointent sur le même objet par jeu de références et donc, inévitablement, si vous modifiez l'objet, vous allez voir le changement depuis les deux variables. Toutefois, il existe un moyen pour créer un nouvel objet depuis un autre :

```
1 >>> ma_liste1 = [1, 2, 3]
2 >>> ma_liste2 = list(ma_liste1) # Cela revient à copier le contenu de ma_liste1
3 >>> ma_liste2.append(4)
4 >>> print(ma_liste2)
5 [1, 2, 3, 4]
6 >>> print(ma_liste1)
7 [1, 2, 3]
8 >>>
```

À la ligne 2, nous avons demandé à Python de créer un nouvel objet basé sur `ma_liste1`. Du coup, les deux variables ne contiennent plus la même référence : elles modifient des objets différents. Vous pouvez utiliser la plupart des constructeurs (c'est le nom qu'on donne à `list` pour créer une liste par exemple) dans ce but. Pour des dictionnaires, utilisez le constructeur `dict` en lui passant en paramètre un dictionnaire déjà construit et vous aurez en retour un dictionnaire, semblable à celui passé en paramètre, mais seulement semblable par le contenu. En fait, il s'agit d'une copie de l'objet, ni plus ni moins.

Pour approcher de plus près les références, vous avez la fonction `id` qui prend en paramètre un objet. Elle renvoie la position de l'objet dans la mémoire Python sous la

forme d'un entier (plutôt grand). Je vous invite à faire quelques tests en passant divers objets en paramètre à cette fonction. Sachez au passage que `is` compare les ID des objets de part et d'autre et c'est pour cette raison que je vous ais mis en garde quant à son utilisation.

```
>>> ma_liste1 = [1, 2]
>>> ma_liste2 = [1, 2]
>>> ma_liste1 == ma_liste2 # On compare le contenu des listes
True
>>> ma_liste1 is ma_liste2 # On compare leur référence
False
>>>
```

Je ne peux que vous encourager à faire des tests avec différents objets. Un petit tour du côté des variables globales ?

Les variables globales

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des **variables globales**.

Cette distinction entre variables locales et variables globales se retrouve dans d'autres langages et on recommande souvent d'éviter de trop les utiliser. Elles peuvent avoir leur utilité, toutefois, puisque le mécanisme existe. D'un point de vue strictement personnel, tant que c'est possible, je ne travaille qu'avec des variables locales (comme nous l'avons fait depuis le début de ce cours) mais il m'arrive de faire appel à des variables globales quand c'est nécessaire ou bien plus pratique. Mais ne tombez pas dans l'extrême non plus, ni dans un sens ni dans l'autre.

Le principe des variables globales

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal. Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

Python va regarder dans les différents espaces : celui de la fonction, celui dans lequel la fonction a été appelée... ainsi de suite jusqu'à mettre la main sur notre variable. S'il la trouve, il va nous donner le plein accès à cette variable dans le corps de la fonction.

Cela signifie que nous pouvons y accéder en lecture (comme c'est le cas sans avoir besoin de la définir comme variable globale) mais aussi en écriture. Une fonction peut donc ainsi changer la valeur d'une variable directement.

Mais assez de théorie, voyons un exemple.

Utiliser concrètement les variables globales

Pour déclarer à Python, dans le corps d'une fonction, que la variable qui sera utilisée doit être considérée comme globale, on utilise le mot-clé `global`. On le place généralement après la définition de la fonction, juste en-dessous de la `docstring`, cela permet de retrouver rapidement les variables globales sans parcourir tout le code (c'est une simple convention). On précise derrière ce mot-clé le nom de la variable à considérer comme globale :

```
>>> i = 4 # Une variable, nommée i, contenant un entier
>>> def inc_i():
...     """Fonction chargée d'incrémenter i de 1"""
...     global i # Python recherche i en dehors de l'espace local de la fonction
...     i += 1
...
>>> i
4
>>> inc_i()
>>> i
5
>>>
```

Si vous ne précisez pas à Python que `i` doit être considérée comme globale, vous ne pourrez pas modifier réellement sa valeur, comme nous l'avons vu plus haut. En précisant `global i`, Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.

J'utilise ce mécanisme quand je travaille sur plusieurs classes et fonctions qui doivent s'échanger des informations d'état par exemple. Il existe d'autres moyens mais vous connaissez celui-ci et, tant que vous maîtrisez bien votre code, il n'est pas plus mauvais qu'un autre.

En résumé

- Les variables locales définies avant l'appel d'une fonction seront accessibles, depuis le corps de la fonction, en lecture seule.
- Une variable locale définie dans une fonction sera supprimée après l'exécution de cette fonction.
- On peut cependant appeler les attributs et méthodes d'un objet pour le modifier durablement.
- Les variables globales se définissent à l'aide du mot-clé `global` suivi du nom de la variable préalablement créée.
- Les variables globales peuvent être modifiées depuis le corps d'une fonction (à utiliser avec prudence).

Chapitre 16

TP : un bon vieux pendu

Difficulté : 

C'est le moment de mettre en pratique ce que vous avez appris. Vous n'aurez pas besoin de tout, bien entendu, mais je vais essayer de vous faire travailler un maximum de choses.

Nous allons donc faire un jeu de pendu plutôt classique. Ce n'est pas bien original mais on va pimenter un peu l'exercice, vous allez voir.



Votre mission

Nous y voilà. Je vais vous préciser un peu la mission, sans quoi on va avoir du mal à s'entendre sur la correction.

Un jeu du pendu

Le premier point de la mission est de réaliser un jeu du pendu. Je rappelle brièvement les règles, au cas où : l'ordinateur choisit un mot au hasard dans une liste, un mot de huit lettres maximum. Le joueur tente de trouver les lettres composant le mot. À chaque coup, il saisit une lettre. Si la lettre figure dans le mot, l'ordinateur affiche le mot avec les lettres déjà trouvées. Celles qui ne le sont pas encore sont remplacées par des étoiles (*). Le joueur a 8 chances. Au delà, il a perdu.

On va compliquer un peu les règles en demandant au joueur de donner son nom, au début de la partie. Cela permettra au programme d'enregistrer son score.

Le score du joueur sera simple à calculer : on prend le score courant (0 si le joueur n'a aucun score déjà enregistré) et, à chaque partie, on lui ajoute le nombre de coups restants comme points de partie. Si, par exemple, il me reste trois coups au moment où je trouve le mot, je gagne trois points.

Par la suite, vous pourrez vous amuser à faire un décompte plus poussé du score, pour l'instant cela suffira bien.

Le côté technique du problème

Le jeu du pendu en lui-même, vous ne devriez avoir aucun problème à le mettre en place. Rappelez-vous que le joueur ne doit donner qu'une seule lettre à la fois et que le programme doit bien vérifier que c'est le cas avant de continuer. Nous allons découper notre programme en trois fichiers :

- Le fichier `donnees.py` qui contiendra les variables nécessaires à notre application (la liste des mots, le nombre de chances autorisées...).
- Le fichier `fonctions.py` qui contiendra les fonctions utiles à notre application. Là, je ne vous fais aucune liste claire, je vous conseille de bien y réfléchir, avec une feuille et un stylo si cela vous aide (Quelles sont les actions de mon programme ? Que puis-je mettre dans des fonctions ?).
- Enfin, notre fichier `pendu.py` qui contiendra notre jeu du pendu.

Gérer les scores

Vous avez, j'espère, une petite idée de comment faire cela... mais je vais quand même clarifier : on va enregistrer dans un fichier de données, que l'on va appeler `scores` (sans aucune extension) les scores du jeu. Ces scores seront sous la forme d'un dictionnaire : en clés, nous aurons les noms des joueurs et en valeurs les scores, sous la forme d'entiers.

Il faut gérer les cas suivants :

– Le fichier n'existe pas. Là, on crée un dictionnaire vide, aucun score n'a été trouvé.
– Le joueur n'est pas dans le dictionnaire. Dans ce cas, on l'ajoute avec un score de `0`.

À vous de jouer

Vous avez l'essentiel. Peut-être pas tout ce dont vous avez besoin, cela dépend de comment vous vous organisez, mais le but est aussi de chercher ! Encore une fois, c'est un exercice pratique, ne sautez pas à la correction tout de suite, cela ne vous apprendra pas grand chose.

Bonne chance !

Correction proposée

Voici la correction que je vous propose. J'espère que vous êtes arrivés à un résultat satisfaisant, même si vous n'avez pas forcément réussi à tout faire. Si votre jeu marche, c'est parfait !

▷ Télécharger les fichiers
Code web : 934163

Voici le code des trois fichiers.

donnees.py

```
"""Ce fichier définit quelques données, sous la forme de
    variables,
utiles au programme pendu"""

# Nombre de coups par partie
nb_coups = 8

# Nom du fichier stockant les scores
nom_fichier_scores = "scores"

# Liste des mots du pendu
liste_mots = [
    "armoire",
    "boucle",
    "buisson",
    "bureau",
    "chaise",
    "carton",
    "couteau",
    "fichier",
    "garage",
```

```
    "glace",
    "journal",
    "kiwi",
    "lampe",
    "liste",
    "montagne",
    "remise",
    "sandale",
    "taxi",
    "vampire",
    "volant",
]
```

fonctions.py

```
"""Ce fichier définit des fonctions utiles pour le programme pendu.

On utilise les données du programme contenues dans donnees.py"""

import os
import pickle
from random import choice

from donnees import *

# Gestion des scores

def recup_scores():
    """Cette fonction récupère les scores enregistrés si le fichier existe.
    Dans tous les cas, on renvoie un dictionnaire,
    soit l'objet dépicklé,
    soit un dictionnaire vide.

    On s'appuie sur nom_fichier_scores défini dans donnees.py"""

    if os.path.exists(nom_fichier_scores): # Le fichier existe
        # On le récupère
        fichier_scores = open(nom_fichier_scores, "rb")
        mon_depickler = pickle.Unpickler(fichier_scores)
        scores = mon_depickler.load()
        fichier_scores.close()
    else: # Le fichier n'existe pas
        scores = {}
    return scores

def enregistrer_scores(scores):
```

```python
    """Cette fonction se charge d'enregistrer les scores dans le fichier
    nom_fichier_scores. Elle reçoit en paramètre le dictionnaire des scores
    à enregistrer"""

    fichier_scores = open(nom_fichier_scores, "wb") # On écrase les anciens scores
    mon_pickler = pickle.Pickler(fichier_scores)
    mon_pickler.dump(scores)
    fichier_scores.close()

# Fonctions gérant les éléments saisis par l'utilisateur

def recup_nom_utilisateur():
    """Fonction chargée de récupérer le nom de l'utilisateur.
    Le nom de l'utilisateur doit être composé de 4 caractères minimum,
    chiffres et lettres exclusivement.

    Si ce nom n'est pas valide, on appelle récursivement la fonction
    pour en obtenir un nouveau"""

    nom_utilisateur = input("Tapez votre nom: ")
    # On met la première lettre en majuscule et les autres en minuscules
    nom_utilisateur = nom_utilisateur.capitalize()
    if not nom_utilisateur.isalnum() or len(nom_utilisateur)<4:
        print("Ce nom est invalide.")
        # On appelle de nouveau la fonction pour avoir un autre nom
        return recup_nom_utilisateur()
    else:
        return nom_utilisateur

def recup_lettre():
"""Cette fonction récupère une lettre saisie par
    l'utilisateur. Si la chaîne récupérée n'est pas une lettre,
    on appelle récursivement la fonction jusqu'à obtenir une lettre"""

    lettre = input("Tapez une lettre: ")
    lettre = lettre.lower()
    if len(lettre)>1 or not lettre.isalpha():
        print("Vous n'avez pas saisi une lettre valide.")
        return recup_lettre()
    else:
        return lettre

```

```python
# Fonctions du jeu de pendu

def choisir_mot():
    """Cette fonction renvoie le mot choisi dans la liste des mots
    liste_mots.

    On utilise la fonction choice du module random (voir l'aide)."""

    return choice(liste_mots)

def recup_mot_masque(mot_complet, lettres_trouvees):
    """Cette fonction renvoie un mot masqué tout ou en partie, en fonction :
    - du mot d'origine (type str)
    - des lettres déjà trouvées (type list)

    On renvoie le mot d'origine avec des * remplaçant les lettres que l'on
    n'a pas encore trouvées."""

    mot_masque = ""
    for lettre in mot_complet:
        if lettre in lettres_trouvees:
            mot_masque += lettre
        else:
            mot_masque += "*"
    return mot_masque
```

pendu.py

```python
"""Ce fichier contient le jeu du pendu.

Il s'appuie sur les fichiers :
- donnees.py
- fonctions.py"""


from donnees import *
from fonctions import *

# On récupère les scores de la partie
scores = recup_scores()

# On récupère un nom d'utilisateur
utilisateur = recup_nom_utilisateur()

# Si l'utilisateur n'a pas encore de score, on l'ajoute
if utilisateur not in scores.keys():
```

```python
    scores[utilisateur] = 0 # 0 point pour commencer

# Notre variable pour savoir quand arrêter la partie
continuer_partie = 'o'

while continuer_partie != 'n':
    print("Joueur {0}: {1} point(s)".format(utilisateur, scores[utilisateur]))
    mot_a_trouver = choisir_mot()
    lettres_trouvees = []
    mot_trouve = recup_mot_masque(mot_a_trouver, lettres_trouvees)
    nb_chances = nb_coups
    while mot_a_trouver!=mot_trouve and nb_chances>0:
        print("Mot à trouver {0} (encore {1} chances)".format(mot_trouve, nb_chances))
        lettre = recup_lettre()
        if lettre in lettres_trouvees: # La lettre a déjà été choisie
            print("Vous avez déjà choisi cette lettre.")
        elif lettre in mot_a_trouver: # La lettre est dans le mot à trouver
            lettres_trouvees.append(lettre)
            print("Bien joué.")
        else:
            nb_chances -= 1
            print("... non, cette lettre ne se trouve pas dans le mot...")
        mot_trouve = recup_mot_masque(mot_a_trouver, lettres_trouvees)

    # A-t-on trouvé le mot ou nos chances sont-elles épuisées ?
    if mot_a_trouver==mot_trouve:
    print("Félicitations ! Vous avez trouvé le mot {0}.".format(mot_a_trouver))
    else:
        print("PENDU !!! Vous avez perdu.")

    # On met à jour le score de l'utilisateur
    scores[utilisateur] += nb_chances

    continuer_partie = input("Souhaitez-vous continuer la partie (O/N) ?")
    continuer_partie = continuer_partie.lower()

# La partie est finie, on enregistre les scores
enregistrer_scores(scores)

# On affiche les scores de l'utilisateur
print("Vous finissez la partie avec {0} points.".format(scores[
```

```
        utilisateur]))
```

Résumé

Dans l'ensemble, je ne pense pas que le code soit très délicat à comprendre. Vous pouvez vous rendre compte à quel point le code du jeu est facile à lire grâce à nos fonctions. On délègue une partie de l'application à nos fonctions qui s'assurent que les choses sont « bien faites ». Si un bug survient, il est plus facile de modifier une fonction que tout un code sans aucune structure.

Par cet exemple, j'espère que vous prendrez bien l'habitude de documenter un maximum vos fichiers et fonctions. C'est réellement un bon réflexe à avoir.



N'oubliez pas la spécification de l'encodage en tête de chaque fichier, ni la mise en pause du programme sous Windows.

Troisième partie

La Programmation Orientée Objet côté développeur

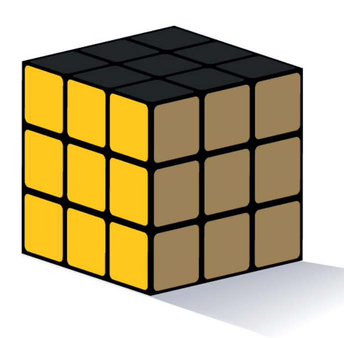
Chapitre 17

Première approche des classes

Difficulté : 

Dans ce chapitre, sans plus attendre, nous allons créer nos premières classes, nos premiers attributs et nos premières méthodes. Nous allons aussi essayer de comprendre les mécanismes de la programmation orientée objet en Python.

Au-delà du mécanisme, l'orienté objet est une véritable philosophie et Python est assez différent des autres langages, en termes de philosophie justement. Restez concentrés, ce langage n'a pas fini de vous étonner !



Les classes, tout un monde

Dans la partie précédente, j'avais brièvement décrit les objets comme des variables pouvant contenir elles-mêmes des fonctions et variables. Nous sommes allés plus loin tout au long de la seconde partie, pour découvrir que nos « fonctions contenues dans nos objets » sont appelées des méthodes. En vérité, je me suis cantonné à une définition « pratique » des objets, alors que derrière la POO (Programmation Orientée Objet) se cache une véritable philosophie.

Pourquoi utiliser des objets ?

Les premiers langages de programmation n'incluaient pas l'orienté objet. Le langage C, pour ne citer que lui, n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C.

Java, un langage apparu à peu près en même temps que Python, définit une philosophie assez différente de celle du C++ : contrairement à ce dernier, le Java exige que tout soit rangé dans des classes. Même l'application standard `Hello World` est contenue dans une classe.

En Python, la liberté est plus grande. Après tout, vous avez pu passer une partie de ce tutoriel sans connaître la façade objet de Python. Et pourtant, le langage Python est totalement orienté objet : *en Python, tout est objet*, vous n'avez pas oublié ? Quand vous croyez utiliser une simple variable, un module, une fonction. . ., ce sont des objets qui se cachent derrière.

Loin de moi l'idée de faire un comparatif entre différents langages. Ce sur quoi je souhaite attirer votre attention, c'est que plusieurs langages intègrent l'orienté objet, chacun avec une philosophie distincte. Autrement dit, si vous avez appris l'orienté objet dans un autre langage, tel que le C++ ou le Java, ne tenez pas pour acquis que vous allez retrouver les même mécanismes et surtout, la même philosophie. Gardez autant que possible l'esprit dégagé de tout préjugé sur la philosophie objet de Python.

Pour l'instant, nous n'avons donc vu qu'un aspect technique de l'objet. J'irais jusqu'à dire que ce qu'on a vu jusqu'ici, ce n'était qu'une façon « un peu plus esthétique » de coder : il est plus simple et plus compréhensible d'écrire `ma_liste.append(5)` que `append_to_list(ma_liste, 5)`. Mais derrière la POO, il n'y a pas qu'un souci esthétique, loin de là.

Choix du modèle

Bon, comme vous vous en souvenez sûrement (du moins, je l'espère), une classe est un peu un modèle suivant lequel on va créer des objets. C'est dans la classe que nous allons définir nos méthodes et attributs, les attributs étant des variables contenues dans notre objet.

Mais qu'allons-nous modéliser ? L'orienté objet est plus qu'utile dès lors que l'on s'en sert pour modéliser, représenter des données un peu plus complexes qu'un simple

nombre, ou qu'une chaîne de caractères. Bien sûr, il existe des classes que Python définit pour nous : les nombres, les chaînes et les listes en font partie. Mais on serait bien limité si on ne pouvait faire ses propres classes.

Pour l'instant, nous allons modéliser... une personne. C'est le premier exemple qui me soit venu à l'esprit, nous verrons bien d'autres exemples avant la fin de la partie.

Convention de nommage

Loin de moi l'idée de compliquer l'exercice mais si on se réfère à la PEP 8[1] de Python , il est préférable d'utiliser pour des noms de classes la convention dite **Camel Case**.

▷ PEP 8 de Python
Code web : 484505

Cette convention n'utilise pas le signe souligné _ pour séparer les mots. Le principe consiste à mettre en majuscule chaque lettre débutant un mot, par exemple : `MaClasse`.

C'est donc cette convention que je vais utiliser pour les noms de classes. Libre à vous d'en changer, encore une fois rien n'est imposé.

Pour définir une nouvelle classe, on utilise le mot-clé `class`.

Sa syntaxe est assez intuitive : `class NomDeLaClasse:`.

N'exécutez pas encore ce code, nous ne savons pas comment définir nos attributs et nos méthodes.

Petit exercice de modélisation : que va-t-on trouver dans les caractéristiques d'une personne ? Beaucoup de choses, vous en conviendrez. On ne va en retenir que quelques-unes : le nom, le prénom, l'âge, le lieu de résidence... allez, cela suffira.

Cela nous fait donc quatre attributs. Ce sont les variables internes à notre objet, qui vont le caractériser. Une personne telle que nous la modélisons sera caractérisée par son nom, son prénom, son âge et son lieu de résidence.

Pour définir les attributs de notre objet, il faut définir un constructeur dans notre classe. Voyons cela de plus près.

Nos premiers attributs

Nous avons défini les attributs qui allaient caractériser notre objet de classe `Personne`. Maintenant, il faut définir dans notre classe une méthode spéciale, appelée un **constructeur**, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

1. Les PEP sont les « Python Enhancement Proposals », c'est à dire les propositions d'amélioration de Python.

Voyons le code, ce sera plus parlant :

```
class Personne: # Définition de notre classe Personne
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self): # Notre méthode constructeur
        """Pour l'instant, on ne va définir qu'un seul attribut
        """
        self.nom = "Dupont"
```

Voyons en détail :

- D'abord, la définition de la classe. Elle est constituée du mot-clé `class`, du nom de la classe et des deux points rituels « : ».
- Une `docstring` commentant la classe. Encore une fois, c'est une excellente habitude à prendre et je vous encourage à le faire systématiquement. Ce pourra être plus qu'utile quand vous vous lancerez dans de grands projets, notamment à plusieurs.
- La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom `__init__`, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés (`__nommethode__`) sont des **méthodes spéciales**. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé `self`.
- Une nouvelle `docstring`. Je ne complique pas inutilement, je précise donc qu'on va simplement définir un seul attribut pour l'instant dans notre constructeur.
- Dans notre constructeur, nous trouvons l'instanciation de notre attribut `nom`. On crée une variable `self.nom` et on lui donne comme valeur `Dupont`. Je vais détailler un peu plus bas ce qui se passe ici.

Avant tout, pour voir le résultat en action, essayons de créer un objet issu de notre classe :

```
>>> bernard = Personne()
>>> bernard
<__main__.Personne object at 0x00B42570>
>>> bernard.nom
'Dupont'
>>>
```

Quand on demande à l'interpréteur d'afficher directement notre objet `bernard`, il nous sort quelque chose d'un peu imbuvable… Bon, l'essentiel est la mention précisant la classe dont l'objet est issu. On peut donc vérifier que c'est bien notre classe `Personne` dont est issu notre objet. On essaye ensuite d'afficher l'attribut `nom` de notre objet `bernard` et on obtient `'Dupont'` (la valeur définie dans notre constructeur). Notez qu'on utilise le point (.), encore et toujours utilisé pour une relation d'appartenance

(nom est un attribut de l'objet bernard). Encore un peu d'explications :

Quand on crée notre objet...

Quand on tape Personne(), on appelle le constructeur de notre classe Personne, d'une façon quelque peu indirecte que je ne détaillerai pas ici. Celui-ci prend en paramètre une variable un peu mystérieuse : self. En fait, il s'agit tout bêtement de notre objet en train de se créer. On écrit dans cet objet l'attribut nom le plus simplement du monde : self.nom = "Dupont". À la fin de l'appel au constructeur, Python renvoie notre objet self modifié, avec notre attribut. On va réceptionner le tout dans notre variable bernard.

Si ce n'est pas très clair, pas de panique ! Vous pouvez vous contenter de vous familiariser avec la syntaxe du constructeur Python, qui sera souvent la même, et laisser l'aspect un peu théorique de côté, pour plus tard. Nous aurons l'occasion d'y revenir avant la fin du chapitre.

Étoffons un peu notre constructeur



Bon, on avait dit quatre attributs, on n'en a fait qu'un. Et puis notre constructeur pourrait éviter de donner les mêmes valeurs par défaut à chaque fois, tout de même !

C'est juste. Dans un premier temps, on va se contenter de définir les autres attributs, le prénom, l'âge, le lieu de résidence. Essayez de le faire, normalement vous ne devriez éprouver aucune difficulté.

Voici le code, au cas où :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self): # Notre méthode constructeur
        """Constructeur de notre classe. Chaque attribut va être instancié
        avec une valeur par défaut... original"""


        self.nom = "Dupont"
        self.prenom = "Jean" # Quelle originalité
        self.age = 33 # Cela n'engage à rien
        self.lieu_residence = "Paris"
```

Cela vous paraît évident ? Encore un petit code d'exemple :

```
1 >>> jean = Personne()
2 >>> jean.nom
3 'Dupont'
4 >>> jean.prenom
5 'Jean'
6 >>> jean.age
7 33
8 >>> jean.lieu_residence
9 'Paris'
10 >>> # Jean déménage..
11 ... jean.lieu_residence = "Berlin"
12 >>> jean.lieu_residence
13 'Berlin'
14 >>>
```

Je sens un courant d'air. . . les habitués de l'objet, une minute.

Cet exemple me paraît assez clair, sur le principe de définition des attributs, accès aux attributs d'un objet créé, modification des attributs d'un objet.

Une toute petite explication en ce qui concerne la ligne 11 : dans beaucoup de tutoriels, on déconseille de modifier un attribut d'instance (un attribut d'un objet) comme on vient de le faire, en faisant simplement `objet.attribut = valeur`. Si vous venez d'un autre langage, vous pourrez avoir entendu parler des accesseurs et mutateurs. Ces concepts sont repris dans certains tutoriels Python, mais ils n'ont pas précisément lieu d'être dans ce langage. Tout cela, je le détaillerai dans le prochain chapitre. Pour l'instant, il vous suffit de savoir que, quand vous voulez modifier un attribut d'un objet, vous écrivez `objet.attribut = nouvelle_valeur`. Nous verrons les cas particuliers plus loin.

Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons. . . le nom et le prénom, pour commencer.

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
```

```
        self.lieu_residence = "Paris"
```

Et en images :

```
>>> bernard = Personne("Micado", "Bernard")
>>> bernard.nom
'Micado'
>>> bernard.prenom
'Bernard'
>>> bernard.age
33
>>>
```

N'oubliez pas que le premier paramètre doit être `self`. En dehors de cela, un constructeur est une fonction plutôt classique : vous pouvez définir des paramètres, par défaut ou non, nommés ou non. Quand vous voudrez créer votre objet, vous appellerez le nom de la classe en passant entre parenthèses les paramètres à utiliser. Faites quelques tests, avec plus ou moins de paramètres, je pense que vous saisirez très rapidement le principe.

Attributs de classe

Dans les exemples que nous avons vus jusqu'à présent, nos attributs sont contenus dans notre objet. Ils sont propres à l'objet : si vous créez plusieurs objets, les attributs `nom`, `prenom`,... de chacun ne seront pas forcément identiques d'un objet à l'autre. Mais on peut aussi définir des attributs dans notre classe. Voyons un exemple :

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incré
        mente à chaque
    fois que l'on crée un objet de ce type"""


    objets_crees = 0 # Le compteur vaut 0 au départ
    def __init__(self):
        """À chaque fois qu'on crée un objet, on incrémente le
            compteur"""
        Compteur.objets_crees += 1
```

On définit notre attribut de classe directement dans le corps de la classe, sous la définition et la `docstring`, avant la définition du constructeur. Quand on veut l'appeler dans le constructeur, on préfixe le nom de l'attribut de classe par le nom de la classe. Et on y accède de cette façon également, en dehors de la classe. Voyez plutôt :

```
>>> Compteur.objets_crees
0
>>> a = Compteur() # On crée un premier objet
>>> Compteur.objets_crees
1
```

```
>>> b = Compteur()
>>> Compteur.objets_crees
2
>>>
```

À chaque fois qu'on crée un objet de type `Compteur`, l'attribut de classe `objets_crees` s'incrémente de 1. Cela peut être utile d'avoir des attributs de classe, quand tous nos objets doivent avoir certaines données identiques. Nous aurons l'occasion d'en reparler par la suite.

Les méthodes, la recette

Les attributs sont des variables propres à notre objet, qui servent à le caractériser. Les méthodes sont plutôt des actions, comme nous l'avons vu dans la partie précédente, agissant sur l'objet. Par exemple, la méthode `append` de la classe `list` permet d'ajouter un élément dans l'objet `list` manipulé.

Pour créer nos premières méthodes, nous allons modéliser… un tableau. Un tableau noir, oui c'est très bien.

Notre tableau va posséder une surface (un attribut) sur laquelle on pourra écrire, que l'on pourra lire et effacer. Pour créer notre classe `TableauNoir` et notre attribut `surface`, vous ne devriez pas avoir de problème :

```python
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut écrire,
    que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié
    est 'surface'"""


    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""
```

Nous avons déjà créé une méthode, aussi vous ne devriez pas être trop surpris par la syntaxe que nous allons voir. Notre constructeur est en effet une méthode, elle en garde la syntaxe. Nous allons donc écrire notre méthode `ecrire` pour commencer.

```python
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut écrire,
    que l'on peut lire et effacer, par jeu de méthodes. L'attribut modifié
    est 'surface'"""


    def __init__(self):
```

```
        """Par défaut, notre surface est vide"""
        self.surface = ""
    def ecrire(self, message_a_ecrire):
        """Méthode permettant d'écrire sur la surface du tableau.
        Si la surface n'est pas vide, on saute une ligne avant de rajouter
        le message à écrire"""


        if self.surface != "":
            self.surface += "\n"
        self.surface += message_a_ecrire
```

Passons aux tests :

```
>>> tab = TableauNoir()
>>> tab.surface
''
>>> tab.ecrire("Coooool ! Ce sont les vacances !")
>>> tab.surface
"Coooool ! Ce sont les vacances !"
>>> tab.ecrire("Joyeux Noël !")
>>> tab.surface
"Coooool ! Ce sont les vacances !\nJoyeux Noël !"
>>> print(tab.surface)
Coooool ! Ce sont les vacances !
Joyeux Noël !
>>>
```

Notre méthode `ecrire` se charge d'écrire sur notre surface, en rajoutant un saut de ligne pour séparer chaque message.

On retrouve ici notre paramètre `self`. Il est temps de voir un peu plus en détail à quoi il sert.

Le paramètre `self`

Dans nos méthodes d'instance, qu'on appelle également des **méthodes d'objet**, on trouve dans la définition ce paramètre `self`. L'heure est venue de comprendre ce qu'il signifie.

Une chose qui a son importance : quand vous créez un nouvel objet, ici un tableau noir, les attributs de l'objet sont propres à l'objet créé. C'est logique : si vous créez plusieurs tableaux noirs, ils ne vont pas tous avoir la même surface. Donc les attributs sont contenus dans l'objet.

En revanche, les méthodes sont contenues dans la classe qui définit notre objet. C'est très important. Quand vous tapez `tab.ecrire(...)`, Python va chercher la méthode `ecrire` non pas dans l'objet `tab`, mais dans la classe `TableauNoir`.

```
>>> tab.ecrire
<bound method TableauNoir.ecrire of <__main__.TableauNoir
    object at 0x00B3F3F0>>
>>> TableauNoir.ecrire
<function ecrire at 0x00BA5810>
>>> help(TableauNoir.ecrire)
Help on function ecrire in module __main__:
ecrire(self, message_a_ecrire)
    Méthode permettant d'écrire sur la surface du tableau.
    Si la surface n'est pas vide, on saute une ligne avant de
        rajouter
    le message à écrire.
>>> TableauNoir.ecrire(tab, "essai")
>>> tab.surface
'essai'
>>>
```

Comme vous le voyez, quand vous tapez `tab.ecrire(...)`, cela revient au même que si vous écrivez `TableauNoir.ecrire(tab, ...)`. Votre paramètre `self`, c'est l'objet qui appelle la méthode. C'est pour cette raison que vous modifiez la surface de l'objet en appelant `self.surface`.

Pour résumer, quand vous devez travailler dans une méthode de l'objet sur l'objet lui-même, vous allez passer par `self`.

Le nom `self` est une très forte convention de nommage. Je vous déconseille de changer ce nom. Certains programmeurs, qui trouvent qu'écrire `self` à chaque fois est excessivement long, l'abrègent en une unique lettre `s`. *Évitez ce raccourci.* De manière générale, évitez de changer le nom. Une méthode d'instance travaille avec le paramètre `self`.



N'est-ce pas effectivement plutôt long de devoir toujours travailler avec self à chaque fois qu'on souhaite faire appel à l'objet ?

Cela peut le sembler, oui. C'est d'ailleurs l'un des reproches qu'on fait au langage Python. Certains langages travaillent implicitement sur les attributs et méthodes d'un objet sans avoir besoin de les appeler spécifiquement. Mais c'est moins clair et cela peut susciter la confusion. En Python, dès qu'on voit `self`, on sait que c'est un attribut ou une méthode interne à l'objet qui va être appelé.

Bon, voyons nos autres méthodes. Nous devons encore coder `lire` qui va se charger d'afficher notre surface et `effacer` qui va effacer le contenu de notre surface. Si vous avez compris ce que je viens d'expliquer, vous devriez écrire ces méthodes sans aucun problème, elles sont très simples. Sinon, n'hésitez pas à relire, jusqu'à ce que le déclic se fasse.

```
class TableauNoir:
    """Classe définissant une surface sur laquelle on peut é
        crire,
```

```
    que l'on peut lire et effacer, par jeu de méthodes. L'
        attribut modifié
    est 'surface'"""


    def __init__(self):
        """Par défaut, notre surface est vide"""
        self.surface = ""
    def ecrire(self, message_a_ecrire):
        """Méthode permettant d'écrire sur la surface du
            tableau.
        Si la surface n'est pas vide, on saute une ligne avant
            de rajouter
        le message à écrire"""


        if self.surface != "":
            self.surface += "\n"
        self.surface += message_a_ecrire
    def lire(self):
        """Cette méthode se charge d'afficher, grâce à print,
        la surface du tableau"""


        print(self.surface)
    def effacer(self):
        """Cette méthode permet d'effacer la surface du tableau
            """
        self.surface = ""
```

Et encore une fois, le code de test :

```
>>> tab = TableauNoir()
>>> tab.lire()
>>> tab.ecrire("Salut tout le monde.")
>>> tab.ecrire("La forme ?")
>>> tab.lire()
Salut tout le monde.
La forme ?
>>> tab.effacer()
>>> tab.lire()
>>>
```

Et voilà ! Avec nos méthodes bien documentées, un petit coup de `help(TableauNoir)` et vous obtenez une belle description de l'utilité de votre classe. C'est très pratique, n'oubliez pas les docstrings.

Méthodes de classe et méthodes statiques

Comme on trouve des attributs propres à la classe, on trouve aussi des méthodes de classe, qui ne travaillent pas sur l'instance `self` mais sur la classe même. C'est un peu plus rare mais cela peut être utile parfois. Notre méthode de classe se définit exactement comme une méthode d'instance, à la différence qu'elle ne prend pas en premier paramètre `self` (l'instance de l'objet) mais `cls` (la classe de l'objet).

En outre, on utilise ensuite une fonction *built-in* de Python pour lui faire comprendre qu'il s'agit d'une méthode de classe, pas d'une méthode d'instance.

```
class Compteur:
    """Cette classe possède un attribut de classe qui s'incrémente à chaque
    fois que l'on crée un objet de ce type"""


    objets_crees = 0 # Le compteur vaut 0 au départ
    def __init__(self):
        """À chaque fois qu'on crée un objet, on incrémente le compteur"""
        Compteur.objets_crees += 1
    def combien(cls):
        """Méthode de classe affichant combien d'objets ont été créés"""
        print("Jusqu'à présent, {} objets ont été créés.".format(
                cls.objets_crees))
    combien = classmethod(combien)
```

Voyons d'abord le résultat :

```
>>> Compteur.combien()
Jusqu'à présent, 0 objets ont été créés.
>>> a = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 1 objets ont été créés.
>>> b = Compteur()
>>> Compteur.combien()
Jusqu'à présent, 2 objets ont été créés.
>>>
```

Une méthode de classe prend en premier paramètre non pas `self` mais `cls`. Ce paramètre contient la classe (ici `Compteur`).

Notez que vous pouvez appeler la méthode de classe depuis un objet instancié sur la classe. Vous auriez par exemple pu écrire `a.combien()`.

Enfin, pour que Python reconnaisse une méthode de classe, il faut appeler la fonction `classmethod` qui prend en paramètre la méthode que l'on veut convertir et renvoie la méthode convertie.

Si vous êtes un peu perdus, retenez la syntaxe de l'exemple. La plupart du temps, vous définirez des méthodes d'instance comme nous l'avons vu plutôt que des méthodes de classe.

On peut également définir des méthodes statiques. Elles sont assez proches des méthodes de classe sauf qu'elles ne prennent aucun premier paramètre, ni `self` ni `cls`. Elles travaillent donc indépendemment de toute donnée, aussi bien contenue dans l'instance de l'objet que dans la classe.

Voici la syntaxe permettant de créer une méthode statique. Je ne veux pas vous surcharger d'informations et je vous laisse faire vos propres tests si cela vous intéresse :

```
class Test:
    """Une classe de test tout simplement"""
    def afficher():
        """Fonction chargée d'afficher quelque chose"""
        print("On affiche la même chose.")
        print("peu importe les données de l'objet ou de la
            classe.")
    afficher = staticmethod(afficher)
```

Si vous vous emmêlez un peu avec les attributs et méthodes de classe, ce n'est pas bien grave. Retenez surtout les attributs et méthodes d'instance, c'est essentiellement sur ceux-ci que je me suis attardé et c'est ceux que vous retrouverez la plupart du temps.



Rappel : les noms de méthodes encadrés par deux soulignés de part et d'autre sont des **méthodes spéciales**. Ne nommez pas vos méthodes ainsi. Nous découvrirons plus tard ces méthodes particulières. Exemple de nom de méthode à éviter : `__mamethode__`.

Un peu d'introspection



Encore de la philosophie ?

Eh bien… le terme d'**introspection**, je le reconnais, fait penser à quelque chose de plutôt abstrait. Pourtant, vous allez très vite comprendre l'idée qui se cache derrière : Python propose plusieurs techniques pour explorer un objet, connaître ses méthodes ou attributs.



Quel est l'intérêt ? Quand on développe une classe, on sait généralement ce qu'il y a dedans, non ?

En effet. L'utilité, à notre niveau, ne saute pas encore aux yeux. Et c'est pour cela que

je ne vais pas trop m'attarder dessus. Si vous ne voyez pas l'intérêt, contentez-vous de garder dans un coin de votre tête les deux techniques que nous allons voir. Arrivera un jour où vous en aurez besoin ! Pour l'heure donc, voyons plutôt l'effet :

La fonction `dir`

La première technique d'introspection que nous allons voir est la fonction `dir`. Elle prend en paramètre un objet et renvoie la liste de ses attributs et méthodes.

```
class Test:
    """Une classe de test tout simplement"""
    def __init__(self):
        """On définit dans le constructeur un unique attribut"""
        self.mon_attribut = "ok"

    def afficher_attribut(self):
        """Méthode affichant l'attribut 'mon_attribut'"""
        print("Mon attribut est {0}.".format(self.mon_attribut))
```

```
>>> # Créons un objet de la classe Test
... un_test = Test()
>>> un_test.afficher_attribut()
Mon attribut est ok.
>>> dir(un_test)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__g
e__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '_
_setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'affich
er_attribut', 'mon_attribut']
>>>
```

La fonction `dir` renvoie une liste comprenant le nom des attributs et méthodes de l'objet qu'on lui passe en paramètre. Vous pouvez remarquer que tout est mélangé, c'est normal : pour Python, les méthodes, les fonctions, les classes, les modules sont des objets. Ce qui différencie en premier lieu une variable d'une fonction, c'est qu'une fonction est exécutable (*callable*). La fonction `dir` se contente de renvoyer tout ce qu'il y a dans l'objet, sans distinction.



Euh, c'est quoi tout cela ? On n'a jamais défini toutes ces méthodes ou attributs !

Non, en effet. Nous verrons plus loin qu'il s'agit de **méthodes spéciales** utiles à Python.

L'attribut spécial `__dict__`

Par défaut, quand vous développez une classe, tous les objets construits depuis cette classe posséderont un attribut spécial `__dict__`. Cet attribut est un dictionnaire qui contient en guise de clés les noms des attributs et, en tant que valeurs, les valeurs des attributs.

Voyez plutôt :

```
>>> un_test = Test()
>>> un_test.__dict__
{'mon_attribut': 'ok'}
>>>
```



Pourquoi « attribut spécial » ?

C'est un attribut un peu particulier car ce n'est pas vous qui le créez, c'est Python. Il est entouré de deux signes soulignés `__` de part et d'autre, ce qui traduit qu'il a une signification pour Python et n'est pas un attribut « standard ». Vous verrez plus loin dans ce cours des **méthodes spéciales** qui reprennent la même syntaxe.



Peut-on modifier ce dictionnaire ?

Vous le pouvez. Sachez qu'en modifiant la valeur de l'attribut, vous modifiez aussi l'attribut dans l'objet.

```
>>> un_test.__dict__["mon_attribut"] = "plus ok"
>>> un_test.afficher_attribut()
Mon attribut est plus ok.
>>>
```

De manière générale, ne faites appel à l'introspection que si vous avez une bonne raison de le faire et évitez ce genre de syntaxe. Il est quand même plus propre d'écrire `objet.attribut = valeur` que `objet.__dict__[nom_attribut] = valeur`.

Nous n'irons pas plus loin dans ce chapitre. Je pense que vous découvrirez dans la suite de ce livre l'utilité des deux méthodes que je vous ai montrées.

En résumé

- On définit une classe en suivant la syntaxe `class NomClasse:`.
- Les méthodes se définissent comme des fonctions, sauf qu'elles se trouvent dans le corps de la classe.
- Les méthodes d'instance prennent en premier paramètre `self`, l'instance de l'objet manipulé.
- On construit une instance de classe en appelant son constructeur, une méthode d'instance appelée `__init__`.
- On définit les attributs d'une instance dans le constructeur de sa classe, en suivant cette syntaxe : `self.nom_attribut = valeur`.

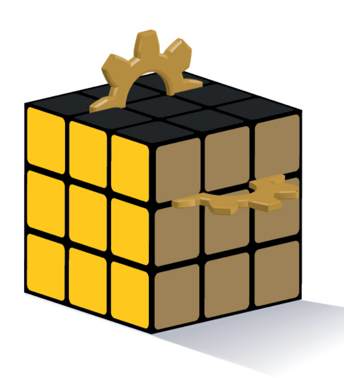
Chapitre 18

Les propriétés

Difficulté : 

Au chapitre précédent, nous avons appris à créer nos premiers attributs et méthodes. Mais nous avons encore assez peu parlé de la philosophie objet. Il existe quelques confusions que je vais tâcher de lever.

Nous allons découvrir dans ce chapitre les propriétés, un concept propre à Python et à quelques autres langages, comme le Ruby. C'est une fonctionnalité qui, à elle seule, change l'approche objet et le principe d'encapsulation.



Qu'est-ce que l'encapsulation ?

L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet. Dans la plupart des langages orientés objet, tels que le C++, le Java ou le PHP, on va considérer que nos attributs d'objets ne doivent pas être accessibles depuis l'extérieur de la classe. Autrement dit, vous n'avez pas le droit de faire, depuis l'extérieur de la classe, `mon_objet.mon_attribut`.



Mais c'est stupide ! Comment fait-on pour accéder aux attributs ?

On va définir des méthodes un peu particulières, appelées des **accesseurs** et **mutateurs**. Les accesseurs donnent accès à l'attribut. Les mutateurs permettent de le modifier. Concrètement, au lieu d'écrire `mon_objet.mon_attribut`, vous allez écrire `mon_objet.get_mon_attribut()` [1]. De la même manière, pour modifier l'attribut écrivez `mon_objet.set_mon_attribut(valeur)` [2] et non pas `mon_objet.mon_attribut = valeur`.



C'est bien tordu tout cela ! Pourquoi ne peut-on pas accéder aux attributs directement, comme on l'a fait au chapitre précédent ?

Ah mais d'abord, je n'ai pas dit que vous ne *pouviez* pas. Vous pouvez très bien accéder aux attributs d'un objet directement, comme on l'a fait au chapitre précédent. Je ne fais ici que résumer le principe d'encapsulation tel qu'on peut le trouver dans d'autres langages. En Python, c'est un peu plus subtil.

Mais pour répondre à la question, il peut être très pratique de sécuriser certaines données de notre objet, par exemple faire en sorte qu'un attribut de notre objet ne soit pas modifiable, ou alors mettre à jour un attribut dès qu'un autre attribut est modifié. Les cas sont multiples et c'est très utile de pouvoir contrôler l'accès en lecture ou en écriture sur certains attributs de notre objet.

L'inconvénient de devoir écrire des accesseurs et mutateurs, comme vous l'aurez sans doute compris, c'est qu'il faut créer deux méthodes pour chaque attribut de notre classe. D'abord, c'est assez lourd. Ensuite, nos méthodes se ressemblent plutôt. Certains environnements de développement proposent, il est vrai, de créer ces accesseurs et mutateurs pour nous, automatiquement. Mais cela ne résout pas vraiment le problème, vous en conviendrez.

Python a une philosophie un peu différente : pour tous les objets dont on n'attend pas une action particulière, on va y accéder directement, comme nous l'avons fait au chapitre précédent. On peut y accéder et les modifier en écrivant simplement `mon_objet.mon_attribut`. Et pour certains, on va créer des propriétés.

1. `get` signifie « récupérer », c'est le préfixe généralement utilisé pour un accesseur.
2. `set` signifie, dans ce contexte, « modifier » ; c'est le préfixe usuel pour un mutateur.

Les propriétés à la casserole

Pour commencer, une petite précision : en C++ ou en Java par exemple, dans la définition de classe, on met en place des principes d'accès qui indiquent si l'attribut (ou le groupe d'attributs) est privé ou public. Pour schématiser, si l'attribut est public, on peut y accéder depuis l'extérieur de la classe et le modifier. S'il est privé, on ne peut pas. On doit passer par des accesseurs ou mutateurs.

En Python, il n'y a pas d'attribut privé. Tout est public. Cela signifie que si vous voulez modifier un attribut depuis l'extérieur de la classe, vous le pouvez. Pour faire respecter l'encapsulation propre au langage, on la fonde sur des conventions que nous allons découvrir un peu plus bas mais surtout sur le bon sens de l'utilisateur de notre classe (à savoir, si j'ai écrit que cet attribut est inaccessible depuis l'extérieur de la classe, je ne vais pas chercher à y accéder depuis l'extérieur de la classe).

Les propriétés sont un moyen transparent de manipuler des attributs d'objet. Elles permettent de dire à Python : « Quand un utilisateur souhaite modifier cet attribut, fais cela ». De cette façon, on peut rendre certains attributs tout à fait inaccessibles depuis l'extérieur de la classe, ou dire qu'un attribut ne sera visible qu'en lecture et non modifiable. Ou encore, on peut faire en sorte que, si on modifie un attribut, Python recalcule la valeur d'un autre attribut de l'objet.

Pour l'utilisateur, c'est absolument transparent : il croit avoir, dans tous les cas, un accès direct à l'attribut. C'est dans la définition de la classe que vous allez préciser que tel ou tel attribut doit être accessible ou modifiable grâce à certaines propriétés.



Mais ces propriétés, c'est quoi ?

Hum... eh bien je pense que pour le comprendre, il vaut mieux les voir en action. Les propriétés sont des objets un peu particuliers de Python. Elles prennent la place d'un attribut et agissent différemment en fonction du contexte dans lequel elles sont appelées. Si on les appelle pour modifier l'attribut, par exemple, elles vont rediriger vers une méthode que nous avons créée, qui gère le cas où « on souhaite modifier l'attribut ». Mais trêve de théorie.

Les propriétés en action

Une propriété ne se crée pas dans le constructeur mais dans le corps de la classe. J'ai dit qu'il s'agissait d'une classe, son nom est `property`. Elle attend quatre paramètres, tous optionnels :

- la méthode donnant accès à l'attribut ;
- la méthode modifiant l'attribut ;
- la méthode appelée quand on souhaite supprimer l'attribut ;
- la méthode appelée quand on demande de l'aide sur l'attribut.

En pratique, on utilise surtout les deux premiers paramètres : ceux définissant les méthodes d'accès et de modification, autrement dit nos accesseur et mutateur d'objet.

Mais j'imagine que ce n'est pas très clair dans votre esprit. Considérez le code suivant, je le détaillerai plus bas comme d'habitude :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom ;
    - son prénom ;
    - son âge ;
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self._lieu_residence = "Paris" # Notez le souligné _ devant le nom
    def _get_lieu_residence(self):
    """Méthode qui sera appelée quand on souhaitera accéder en lecture
        à l'attribut 'lieu_residence'"""


        print("On accède à l'attribut lieu_residence !")
        return self._lieu_residence
    def _set_lieu_residence(self, nouvelle_residence):
        """Méthode appelée quand on souhaite modifier le lieu de résidence"""
        print("Attention, il semble que {} déménage à {}.".format( \
                self.prenom, nouvelle_residence))
        self._lieu_residence = nouvelle_residence
    # On va dire à Python que notre attribut lieu_residence pointe vers une
    # propriété
    lieu_residence = property(_get_lieu_residence, _set_lieu_residence)
```

Vous devriez (j'espère) reconnaître la syntaxe générale de la classe. En revanche, au niveau du lieu de résidence, les choses changent un peu :

– Tout d'abord, dans le constructeur, on ne crée pas un attribut `self.lieu_residence` mais `self._lieu_residence`. Il n'y a qu'un petit caractère de différence, le signe souligné _ placé en tête du nom de l'attribut. Et pourtant, ce signe change beaucoup de choses. La convention veut qu'on n'accède pas, depuis l'extérieur de la classe, à un attribut commençant par un souligné _. C'est une convention, rien ne vous l'interdit… sauf, encore une fois, le bon sens.

– On définit une première méthode, commençant elle aussi par un souligné _, nommée `_get_lieu_residence`. C'est la même règle que pour les attributs : on n'accède pas, depuis l'extérieur de la classe, à une méthode commençant par un souligné _. Si vous avez compris ma petite explication sur les accesseurs et mutateurs, vous devriez comprendre rapidement à quoi sert cette méthode : elle se contente de renvoyer le lieu de résidence. Là encore, l'attribut manipulé n'est pas `lieu_residence` mais `_lieu_residence`. Comme on est dans la classe, on a le droit de le manipuler.
– La seconde méthode a la forme d'un mutateur. Elle se nomme `_set_lieu_residence` et doit donc aussi être inaccessible depuis l'extérieur de la classe. À la différence de l'accesseur, elle prend un paramètre : le nouveau lieu de résidence. En effet, c'est une méthode qui doit être appelée quand on cherche à modifier le lieu de résidence, il lui faut donc le nouveau lieu de résidence qu'on souhaite voir affecté à l'objet.
– Enfin, la dernière ligne de la classe est très intéressante. Il s'agit de la définition d'une propriété. On lui dit que l'attribut `lieu_residence` (cette fois, sans signe souligné _) doit être une propriété. On définit dans notre propriété, dans l'ordre, la méthode d'accès (l'accesseur) et celle de modification (le mutateur).

Quand on veut accéder à `objet.lieu_residence`, Python tombe sur une propriété redirigeant vers la méthode `_get_lieu_residence`. Quand on souhaite modifier la valeur de l'attribut, en écrivant `objet.lieu_residence = valeur`, Python appelle la méthode `_set_lieu_residence` en lui passant en paramètre la nouvelle valeur.

Ce n'est pas clair ? Voyez cet exemple :

```
>>> jean = Personne("Micado", "Jean")
>>> jean.nom
'Micado'
>>> jean.prenom
'Jean'
>>> jean.age
33
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Paris'
>>> jean.lieu_residence = "Berlin"
Attention, il semble que Jean déménage à Berlin.
>>> jean.lieu_residence
On accède à l'attribut lieu_residence !
'Berlin'
>>>
```

Notre accesseur et notre mutateur se contentent d'afficher un message, pour bien qu'on se rende compte que ce sont eux qui sont appelés quand on souhaite manipuler l'attribut `lieu_residence`. Vous pouvez aussi ne définir qu'un accesseur, dans ce cas l'attribut ne pourra pas être modifié.

Il est aussi possible de définir, en troisième position du constructeur `property`, une méthode qui sera appelée quand on fera `del objet.lieu_residence` et, en quatrième position, une méthode qui sera appelée quand on fera `help(objet.lieu_residence)`.

Ces deux dernières fonctionnalités sont un peu moins utilisées mais elles existent.

Voilà, vous connaissez à présent la syntaxe pour créer des propriétés. Entraînez-vous, ce n'est pas toujours évident au début. C'est un concept très puissant, il serait dommage de passer à côté.

Résumons le principe d'encapsulation en Python

Dans cette section, je vais condenser un peu tout le chapitre. Nous avons vu qu'en Python, quand on souhaite accéder à un attribut d'un objet, on écrit tout bêtement `objet.attribut`. Par contre, on doit éviter d'accéder ainsi à des attributs ou des méthodes commençant par un signe souligné _, question de convention. Si par hasard une action particulière doit être menée quand on accède à un attribut, pour le lire tout simplement, pour le modifier, le supprimer. . ., on fait appel à des propriétés. Pour l'utilisateur de la classe, cela revient au même : il écrit toujours `objet.attribut`. Mais dans la définition de notre classe, nous faisons en sorte que l'attribut visé soit une propriété avec certaines méthodes, accesseur, mutateur ou autres, qui définissent ce que Python doit faire quand on souhaite lire, modifier, supprimer l'attribut.

Avec ce concept, on perd beaucoup moins de temps. On ne fait pas systématiquement un accesseur et un mutateur pour chaque attribut et le code est bien plus lisible. C'est autant de gagné.

Certaines classes ont besoin qu'un traitement récurrent soit effectué sur leurs attributs. Par exemple, quand je souhaite modifier un attribut de l'objet (n'importe quel attribut), l'objet doit être enregistré dans un fichier. Dans ce cas, on n'utilisera pas les propriétés, qui sont plus utiles pour des cas particuliers, mais plutôt des méthodes spéciales, que nous découvrirons au prochain chapitre.

En résumé

- Les propriétés permettent de contrôler l'accès à certains attributs d'une instance.
- Elles se définissent dans le corps de la classe en suivant cette syntaxe : `nom_propriete = propriete(methode_accesseur, methode_mutateur, methode_suppression, methode_aide)`.
- On y fait appel ensuite en écrivant `objet.nom_propriete` comme pour n'importe quel attribut.
- Si l'on souhaite juste lire l'attribut, c'est la méthode définie comme accesseur qui est appelée.
- Si l'on souhaite modifier l'attribut, c'est la méthode mutateur, si elle est définie, qui est appelée.
- Chacun des paramètres à passer à `property` est optionnel.

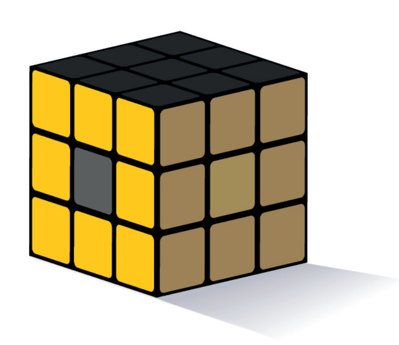
Chapitre 19

Les méthodes spéciales

Difficulté : 

Les méthodes spéciales sont des méthodes d'instance que Python reconnaît et sait utiliser, dans certains contextes. Elles peuvent servir à indiquer à Python ce qu'il doit faire quand il se retrouve devant une expression comme `mon_objet1 + mon_objet2`, voire `mon_objet[indice]`. Et, encore plus fort, elles contrôlent la façon dont un objet se crée, ainsi que l'accès à ses attributs.

Bref, encore une fonctionnalité puissante et utile du langage, que je vous invite à découvrir. Prenez note du fait que je ne peux pas expliquer dans ce chapitre la totalité des méthodes spéciales. Il y en a qui ne sont pas de notre niveau, il y en a sur lesquelles je passerai plus vite que d'autres. En cas de doute, ou si vous êtes curieux, je vous encourage d'autant plus à aller faire un tour sur le site officiel de Python.



Édition de l'objet et accès aux attributs

Vous avez déjà vu, dès le début de cette troisième partie, un exemple de **méthode spéciale**. Pour ceux qui ont la mémoire courte, il s'agit de notre constructeur. Une méthode spéciale, en Python, voit son nom entouré de part et d'autre par deux signes « souligné » _. Le nom d'une méthode spéciale prend donc la forme : `__methodespeciale__`.

Pour commencer, nous allons voir les méthodes qui travaillent directement sur l'objet. Nous verrons ensuite, plus spécifiquement, les méthodes qui permettent d'accéder aux attributs.

Édition de l'objet

Les méthodes que nous allons voir permettent de travailler sur l'objet. Elles interviennent au moment de le créer et au moment de le supprimer. La première, vous devriez la reconnaître : c'est notre constructeur. Elle s'appelle `__init__`, prend un nombre variable d'arguments et permet de contrôler la création de nos attributs.

```
class Exemple:
    """Un petit exemple de classe"""
    def __init__(self, nom):
        """Exemple de constructeur"""
        self.nom = nom
        self.autre_attribut = "une valeur"
```

Pour créer notre objet, nous utilisons le nom de la classe et nous passons, entre parenthèses, les informations qu'attend notre constructeur :

```
mon_objet = Exemple("un premier exemple")
```

J'ai un peu simplifié ce qui se passe mais, pour l'instant, c'est tout ce qu'il vous faut retenir. Comme vous pouvez le voir, à partir du moment où l'objet est créé, on peut accéder à ses attributs grâce à `mon_objet.nom_attribut` et exécuter ses méthodes grâce à `mon_objet.nom_methode(...)`.

Il existe également une autre méthode, `__del__`, qui va être appelée au moment de la destruction de l'objet.



La destruction ? Quand un objet se détruit-il ?

Bonne question. Il y a plusieurs cas : d'abord, quand vous voulez le supprimer explicitement, grâce au mot-clé `del` (`del mon_objet`). Ensuite, si l'espace de noms contenant l'objet est détruit, l'objet l'est également. Par exemple, si vous instanciez l'objet dans le corps d'une fonction : à la fin de l'appel à la fonction, la méthode `__del__` de l'objet sera appelée. Enfin, si votre objet résiste envers et contre tout pendant l'exécution du programme, il sera supprimé à la fin de l'exécution.

```
def __del__(self):
        """Méthode appelée quand l'objet est supprimé"""
        print("C'est la fin ! On me supprime !")
```



À quoi cela peut-il bien servir, de contrôler la destruction d'un objet ?

Souvent, à rien. Python s'en sort comme un grand garçon, il n'a pas besoin d'aide. Parfois, on peut vouloir récupérer des informations d'état sur l'objet au moment de sa suppression. Mais ce n'est qu'un exemple : les méthodes spéciales sont un moyen d'exécuter des actions personnalisées sur certains objets, dans un cas précis. Si l'utilité ne saute pas aux yeux, vous pourrez en trouver une un beau jour, en codant votre projet.

Souvenez-vous que si vous ne définissez pas de méthode spéciale pour telle ou telle action, Python aura un comportement par défaut dans le contexte où cette méthode est appelée. Écrire une méthode spéciale permet de modifier ce comportement par défaut. Dans l'absolu, vous n'êtes même pas obligés d'écrire un constructeur.

Représentation de l'objet

Nous allons voir deux méthodes spéciales qui permettent de contrôler comment l'objet est représenté et affiché. Vous avez sûrement déjà pu constater que, quand on instancie des objets issus de nos propres classes, si on essaye de les afficher directement dans l'interpréteur ou grâce à `print`, on obtient quelque chose d'assez laid :

```
<__main__.XXX object at 0x00B46A70>
```

On a certes les informations utiles, mais pas forcément celles qu'on veut, et l'ensemble n'est pas magnifique, il faut bien le reconnaître.

La première méthode permettant de remédier à cet état de fait est `__repr__`. Elle affecte la façon dont est affiché l'objet quand on tape directement son nom. On la redéfinit quand on souhaite faciliter le **debug** sur certains objets :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
    def __repr__(self):
        """Quand on entre notre objet dans l'interpréteur"""
        return "Personne: nom({}), prénom({}), âge({})".format(
                self.nom, self.prenom, self.age)
```

Et le résultat en images :

```
>>> p1 = Personne("Micado", "Jean")
>>> p1
Personne: nom(Micado), prénom(Jean), âge(33)
>>>
```

Comme vous le voyez, la méthode `__repr__` ne prend aucun paramètre (sauf, bien entendu, `self`) et renvoie une chaîne de caractères : la chaîne à afficher quand on entre l'objet directement dans l'interpréteur.

On peut également obtenir cette chaîne grâce à la fonction `repr`, qui se contente d'appeler la méthode spéciale `__repr__` de l'objet passé en paramètre :

```
>>> p1 = Personne("Micado", "Jean")
>>> repr(p1)
'Personne: nom(Micado), prénom(Jean), âge(33)'
>>>
```

Il existe une seconde méthode spéciale, `__str__`, spécialement utilisée pour afficher l'objet avec `print`. Par défaut, si aucune méthode `__str__` n'est définie, Python appelle la méthode `__repr__` de l'objet. La méthode `__str__` est également appelée si vous désirez convertir votre objet en chaîne avec le constructeur `str`.

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
    def __str__(self):
        """Méthode permettant d'afficher plus joliment notre
           objet"""
        return "{} {}, âgé de {} ans".format(
                self.prenom, self.nom, self.age)
```

Et en pratique :

```
>>> p1 = Personne("Micado", "Jean")
>>> print(p1)
Jean Micado, âgé de 33 ans
>>> chaine = str(p1)
>>> chaine
'Jean Micado, âgé de 33 ans'
>>>
```

Accès aux attributs de notre objet

Nous allons découvrir trois méthodes permettant de définir comment accéder à nos attributs et les modifier.

La méthode `__getattr__`

La méthode spéciale `__getattr__` permet de définir une méthode d'accès à nos attributs plus large que celle que Python propose par défaut. En fait, cette méthode est appelée quand vous tapez `objet.attribut` (non pas pour modifier l'attribut mais simplement pour y accéder). Python recherche l'attribut et, *s'il ne le trouve pas dans l'objet* et si une méthode `__getattr__` existe, il va l'appeler en lui passant en paramètre le nom de l'attribut recherché, sous la forme d'une chaîne de caractères.

Un petit exemple ?

```
>>> class Protege:
...     """Classe possédant une méthode particulière d'accès à ses attributs :
...     Si l'attribut n'est pas trouvé, on affiche une alerte et renvoie None"""
...
...
...     def __init__(self):
...         """On crée quelques attributs par défaut"""
...         self.a = 1
...         self.b = 2
...         self.c = 3
...     def __getattr__(self, nom):
...         """Si Python ne trouve pas l'attribut nommé nom, il appelle
...         cette méthode. On affiche une alerte"""
...
...
...         print("Alerte ! Il n'y a pas d'attribut {} ici !".format(nom))
...
>>> pro = Protege()
>>> pro.a
1
>>> pro.c
3
>>> pro.e
Alerte ! Il n'y a pas d'attribut e ici !
>>>
```

Vous comprenez le principe ? Si l'attribut auquel on souhaite accéder existe, notre méthode n'est pas appelée. En revanche, si l'attribut n'existe pas, notre méthode `__getattr__` est appelée. On lui passe en paramètre le nom de l'attribut auquel Py-

thon essaye d'accéder. Ici, on se contente d'afficher une alerte. Mais on pourrait tout aussi bien rediriger vers un autre attribut. Par exemple, si on essaye d'accéder à un attribut qui n'existe pas, on redirige vers `self.c`. Je vous laisse faire l'essai, cela n'a rien de difficile.

La méthode `__setattr__`

Cette méthode définit l'accès à un attribut destiné à être modifié. Si vous écrivez `objet.nom_attribut = nouvelle_valeur`, la méthode spéciale `__setattr__` sera appelée ainsi : `objet.__setattr__("nom_attribut", nouvelle_valeur)`. Là encore, le nom de l'attribut recherché est passé sous la forme d'une chaîne de caractères. Cette méthode permet de déclencher une action dès qu'un attribut est modifié, par exemple enregistrer l'objet :

```
1 def __setattr__(self, nom_attr, val_attr):
2         """Méthode appelée quand on fait objet.nom_attr = val_attr.
3         On se charge d'enregistrer l'objet"""
4 
5 
6         object.__setattr__(self, nom_attr, val_attr)
7         self.enregistrer()
```

Une explication s'impose concernant la ligne **6**, je pense. Je vais faire de mon mieux, sachant que j'expliquerai bien plus en détail, au prochain chapitre, le concept d'héritage. Pour l'instant, il vous suffit de savoir que toutes les classes que nous créons sont héritées de la classe `object`. Cela veut dire essentiellement qu'elles reprennent les mêmes méthodes. La classe `object` est définie par Python. Je disais plus haut que, si vous ne définissiez pas une certaine méthode spéciale, Python avait un comportement par défaut : ce comportement est défini par la classe `object`.

La plupart des méthodes spéciales sont déclarées dans `object`. Si vous faites par exemple `objet.attribut = valeur` sans avoir défini de méthode `__setattr__` dans votre classe, c'est la méthode `__setattr__` de la classe `object` qui sera appelée.

Mais si vous redéfinissez la méthode `__setattr__` dans votre classe, la méthode appelée sera alors celle que vous définissez, et non celle de `object`. Oui mais... vous ne savez pas comment Python fait, réellement, pour modifier la valeur d'un attribut. Le mécanisme derrière la méthode vous est inconnu.

Si vous essayez, dans la méthode `__setattr__`, de faire `self.attribut = valeur`, vous allez créer une jolie erreur : Python va vouloir modifier un attribut, il appelle la méthode `__setattr__` de la classe que vous avez définie, il tombe dans cette méthode sur une nouvelle affectation d'attribut, il appelle donc de nouveau `__setattr__`... et tout cela, jusqu'à l'infini ou presque. Python met en place une protection pour éviter qu'une méthode ne s'appelle elle-même à l'infini, mais cela ne règle pas le problème.

Tout cela pour dire que, dans votre méthode `__setattr__`, vous ne pouvez pas modifier d'attribut de la façon que vous connaissez. Si vous le faites, `__setattr__` appellera `__setattr__` qui appellera `__setattr__`... à l'infini. Donc si on souhaite modifier un

attribut, on va se référer à la méthode `__setattr__` définie dans la classe `object`, la classe mère dont toutes nos classes héritent.

Si toutes ces explications vous ont paru plutôt dures, ne vous en faites pas trop : je détaillerai au prochain chapitre ce qu'est l'héritage, vous comprendrez sûrement mieux à ce moment.

La méthode `__delattr__`

Cette méthode spéciale est appelée quand on souhaite supprimer un attribut de l'objet, en faisant `del objet.attribut` par exemple. Elle prend en paramètre, outre `self`, le nom de l'attribut que l'on souhaite supprimer. Voici un exemple d'une classe dont on ne peut supprimer aucun attribut :

```
def __delattr__(self, nom_attr):
        """On ne peut supprimer d'attribut, on lève l'exception
        AttributeError"""

        raise AttributeError("Vous ne pouvez supprimer aucun
            attribut de cette classe")
```

Là encore, si vous voulez supprimer un attribut, n'utilisez pas dans votre méthode `del self.attribut`. Sinon, vous risquez de mettre Python très en colère ! Passez par `object.__delattr__` qui sait mieux que nous comment tout cela fonctionne.

Un petit bonus

Voici quelques fonctions qui font à peu près ce que nous avons fait mais en utilisant des chaînes de caractères pour les noms d'attributs. Vous pourrez en avoir l'usage :

```
objet = MaClasse() # On crée une instance de notre classe
getattr(objet, "nom") # Semblable à objet.nom
setattr(objet, "nom", val) # = objet.nom = val ou objet.
    __setattr__("nom", val)
delattr(objet, "nom") # = del objet.nom ou objet.__delattr__("
    nom")
hasattr(objet, "nom") # Renvoie True si l'attribut "nom" existe
    , False sinon
```

Peut-être ne voyez-vous pas trop l'intérêt de ces fonctions qui prennent toutes, en premier paramètre, l'objet sur lequel travailler et en second le nom de l'attribut (sous la forme d'une chaîne). Toutefois, cela peut être très pratique parfois de travailler avec des chaînes de caractères plutôt qu'avec des noms d'attributs. D'ailleurs, c'est un peu ce que nous venons de faire, dans nos redéfinitions de méthodes accédant aux attributs.

Là encore, si l'intérêt ne saute pas aux yeux, laissez ces fonctions de côté. Vous pourrez les retrouver par la suite.

Les méthodes de conteneur

Nous allons commencer à travailler sur ce que l'on appelle la **surcharge d'opérateurs**. Il s'agit assez simplement d'expliquer à Python quoi faire quand on utilise tel ou tel opérateur. Nous allons ici voir quatre méthodes spéciales qui interviennent quand on travaille sur des objets conteneurs.

Accès aux éléments d'un conteneur

Les objets conteneurs, j'espère que vous vous en souvenez, ce sont les chaînes de caractères, les listes et les dictionnaires, entre autres. Tous ont un point commun : ils contiennent d'autres objets, auxquels on peut accéder grâce à l'opérateur `[]`.

Les trois premières méthodes que nous allons voir sont `__getitem__`, `__setitem__` et `__delitem__`. Elles servent respectivement à définir quoi faire quand on écrit :

- `objet[index]` ;
- `objet[index] = valeur` ;
- `del objet[index]` ;

Pour cet exemple, nous allons voir une classe enveloppe de dictionnaire. Les classes enveloppes sont des classes qui ressemblent à d'autres classes mais n'en sont pas réellement. Cela vous avance ?

Nous allons créer une classe que nous allons appeler `ZDict`. Elle va posséder un attribut auquel on ne devra pas accéder de l'extérieur de la classe, un dictionnaire que nous appellerons `_dictionnaire`. Quand on créera un objet de type `ZDict` et qu'on voudra faire `objet[index]`, à l'intérieur de la classe on fera `self._dictionnaire[index]`. En réalité, notre classe fera semblant d'être un dictionnaire, elle réagira de la même manière, mais elle n'en sera pas réellement un.

```
class ZDict:
    """Classe enveloppe d'un dictionnaire"""
    def __init__(self):
        """Notre classe n'accepte aucun paramètre"""
        self._dictionnaire = {}
    def __getitem__(self, index):
        """Cette méthode spéciale est appelée quand on fait objet[index]
        Elle redirige vers self._dictionnaire[index]"""

        return self._dictionnaire[index]
    def __setitem__(self, index, valeur):
        """Cette méthode est appelée quand on écrit objet[index] = valeur
        On redirige vers self._dictionnaire[index] = valeur"""

        self._dictionnaire[index] = valeur
```

Vous avez un exemple d'utilisation des deux méthodes `__getitem__` et `__setitem__`

qui, je pense, est assez clair. Pour `__delitem__`, je crois que c'est assez évident, elle ne prend qu'un seul paramètre qui est l'index que l'on souhaite supprimer. Vous pouvez étendre cet exemple avec d'autres méthodes que nous avons vues plus haut, notamment `__repr__` et `__str__`. N'hésitez pas, entraînez-vous, tout cela peut vous servir.

La méthode spéciale derrière le mot-clé `in`

Il existe une quatrième méthode, appelée `__contains__`, qui est utilisée quand on souhaite savoir si un objet se trouve dans un conteneur.

Exemple classique :

```
ma_liste = [1, 2, 3, 4, 5]
8 in ma_liste # Revient au même que ...
ma_liste.__contains__(8)
```

Ainsi, si vous voulez que votre classe enveloppe puisse utiliser le mot-clé `in` comme une liste ou un dictionnaire, vous devez redéfinir cette méthode `__contains__` qui prend en paramètre, outre `self`, l'objet qui nous intéresse. Si l'objet est dans le conteneur, on doit renvoyer `True` ; sinon `False`.

Je vous laisse redéfinir cette méthode, vous avez toutes les indications nécessaires.

Connaître la taille d'un conteneur

Il existe enfin une méthode spéciale `__len__`, appelée quand on souhaite connaître la taille d'un objet conteneur, grâce à la fonction `len`.

`len(objet)` équivaut à `objet.__len__()`. Cette méthode spéciale ne prend aucun paramètre et renvoie une taille sous la forme d'un entier. Là encore, je vous laisse faire l'essai.

Les méthodes mathématiques

Pour cette section, nous allons continuer à voir les méthodes spéciales permettant la surcharge d'opérateurs mathématiques, comme `+`, `-`, `*` et j'en passe.

Ce qu'il faut savoir

Pour cette section, nous allons utiliser un nouvel exemple, une classe capable de contenir des durées. Ces durées seront contenues sous la forme d'un nombre de minutes et un nombre de secondes.

Voici le corps de la classe, gardez-le sous la main :

```
class Duree:
    """Classe contenant des durées sous la forme d'un nombre de
        minutes
```

```
    et de secondes"""

    def __init__(self, min=0, sec=0):
        """Constructeur de la classe"""
        self.min = min # Nombre de minutes
        self.sec = sec # Nombre de secondes
    def __str__(self):
        """Affichage un peu plus joli de nos objets"""
        return "{0:02}:{1:02}".format(self.min, self.sec)
```

On définit simplement deux attributs contenant notre nombre de minutes et notre nombre de secondes, ainsi qu'une méthode pour afficher tout cela un peu mieux. Si vous vous interrogez sur l'utilisation de la méthode `format` dans la méthode `__str__`, sachez simplement que le but est de voir la durée sous la forme `MM:SS` ; pour plus d'informations sur le formatage des chaînes, vous pouvez consulter le code web suivant :

▷ Documentation de Python
Code web : 973909

Créons un premier objet `Duree` que nous appelons `d1`.

```
>>> d1 = Duree(3, 5)
>>> print(d1)
03:05
>>>
```

Si vous essayez de faire `d1 + 4`, par exemple, vous allez obtenir une erreur. Python ne sait pas comment additionner un type `Duree` et un `int`. Il ne sait même pas comment ajouter deux durées ! Nous allons donc lui expliquer.

La méthode spéciale à redéfinir est `__add__`. Elle prend en paramètre l'objet que l'on souhaite ajouter. Voici deux lignes de code qui reviennent au même :

```
d1 + 4
d1.__add__(4)
```

Comme vous le voyez, quand vous utilisez le symbole + ainsi, c'est en fait la méthode `__add__` de l'objet `Duree` qui est appelée. Elle prend en paramètre l'objet que l'on souhaite ajouter, peu importe le type de l'objet en question. Et elle doit renvoyer un objet exploitable, ici il serait plus logique que ce soit une nouvelle durée.

Si vous devez faire différentes actions en fonction du type de l'objet à ajouter, testez le résultat de `type(objet_a_ajouter)`.

```
def __add__(self, objet_a_ajouter):
        """L'objet à ajouter est un entier, le nombre de
            secondes"""
        nouvelle_duree = Duree()
        # On va copier self dans l'objet créé pour avoir la mê
            me durée
        nouvelle_duree.min = self.min
        nouvelle_duree.sec = self.sec
```

```
        # On ajoute la durée
        nouvelle_duree.sec += objet_a_ajouter
        # Si le nombre de secondes >= 60
        if nouvelle_duree.sec >= 60:
            nouvelle_duree.min += nouvelle_duree.sec // 60
            nouvelle_duree.sec = nouvelle_duree.sec % 60
        # On renvoie la nouvelle durée
        return nouvelle_duree
```

Prenez le temps de comprendre le mécanisme et le petit calcul pour vous assurer d'avoir une durée cohérente. D'abord, on crée une nouvelle durée qui est l'équivalent de la durée contenue dans `self`. On l'augmente du nombre de secondes à ajouter et on s'assure que le temps est cohérent (le nombre de secondes n'atteint pas 60). Si le temps n'est pas cohérent, on le corrige. On renvoie enfin notre nouvel objet modifié. Voici un petit code qui montre comment utiliser notre méthode :

```
>>> d1 = Duree(12, 8)
>>> print(d1)
12:08
>>> d2 = d1 + 54 # d1 + 54 secondes
>>> print(d2)
13:02
>>>
```

Pour mieux comprendre, remplacez `d2 = d1 + 54` par `d2 = d1.__add__(54)` : cela revient au même. Ce remplacement ne sert qu'à bien comprendre le mécanisme. Il va de soi que ces méthodes spéciales ne sont pas à appeler directement depuis l'extérieur de la classe, les opérateurs n'ont pas été inventés pour rien.

Sachez que sur le même modèle, il existe les méthodes :

- `__sub__` : surcharge de l'opérateur `-` ;
- `__mul__` : surcharge de l'opérateur `*` ;
- `__truediv__` : surcharge de l'opérateur `/` ;
- `__floordiv__` : surcharge de l'opérateur `//` (division entière) ;
- `__mod__` : surcharge de l'opérateur `%` (modulo) ;
- `__pow__` : surcharge de l'opérateur `**` (puissance) ;
- ...

Il y en a d'autres que vous pouvez consulter grâce au code web suivant.

▷ Site officiel de Python
Code web : 338274

Tout dépend du sens

Vous l'avez peut-être remarqué, et c'est assez logique si vous avez suivi mes explications, mais écrire `objet1 + objet2` ne revient pas au même qu'écrire `objet2 + objet1` si les deux objets ont des types différents.

En effet, suivant le cas, c'est la méthode `__add__` de l'un ou l'autre des objets qui est appelée.

Cela signifie que, lorsqu'on utilise la classe `Duree`, si on écrit `d1 + 4` cela fonctionne, alors que `4 + d1` ne marche pas. En effet, la class `int` ne sait pas quoi faire de votre objet `Duree`.

Il existe cependant une panoplie de méthodes spéciales pour faire le travail de `__add__` si vous écrivez l'opération dans l'autre sens. Il suffit de préfixer le nom des méthodes spéciales par un `r`.

```
def __radd__(self, objet_a_ajouter):
        """Cette méthode est appelée si on écrit 4 + objet et que
        le premier objet (4 dans cet exemple) ne sait pas comment ajouter
        le second. On se contente de rediriger sur __add__ puisque,
        ici, cela revient au même : l'opération doit avoir le même résultat,
        posée dans un sens ou dans l'autre"""

        return self + objet_a_ajouter
```

À présent, on peut écrire `4 + d1`, cela revient au même que `d1 + 4`.

N'hésitez pas à relire ces exemples s'ils vous paraissent peu clairs.

D'autres opérateurs

Il est également possible de surcharger les opérateurs `+=`, `-=`, etc. On préfixe cette fois-ci les noms de méthode que nous avons vus par un `i`.

Exemple de méthode `__iadd__` pour notre classe `Duree` :

```
def __iadd__(self, objet_a_ajouter):
        """L'objet à ajouter est un entier, le nombre de secondes"""
        # On travaille directement sur self cette fois
        # On ajoute la durée
        self.sec += objet_a_ajouter
        # Si le nombre de secondes >= 60
        if self.sec >= 60:
            self.min += self.sec // 60
            self.sec = self.sec % 60
        # On renvoie self
        return self
```

Et en images :

```
>>> d1 = Duree(8, 5)
>>> d1 += 128
```

```
>>> print(d1)
10:13
>>>
```

Je ne peux que vous encourager à faire des tests, pour être bien sûrs de comprendre le mécanisme. Je vous ai donné ici une façon de faire en la commentant mais, si vous ne pratiquez pas ou n'essayez pas par vous-mêmes, vous n'allez pas la retenir et vous n'allez pas forcément comprendre la logique.

Les méthodes de comparaison

Pour finir, nous allons voir la surcharge des opérateurs de comparaison que vous connaissez depuis quelque temps maintenant : `==`, `!=`, `<`, `>`, `<=`, `>=`.

Ces méthodes sont donc appelées si vous tentez de comparer deux objets entre eux. Comment Python sait-il que 3 est inférieur à 18 ? Une méthode spéciale de la classe `int` le permet, en simplifiant. Donc si vous voulez comparer des durées, par exemple, vous allez devoir redéfinir certaines méthodes que je vais présenter plus bas. Elles devront prendre en paramètre l'objet à comparer à `self`, et doivent renvoyer un booléen (`True` ou `False`).

Je vais me contenter de vous faire un petit tableau récapitulatif des méthodes à redéfinir pour comparer deux objets entre eux :

Opérateur	Méthode spéciale	Résumé
`==`	`def __eq__(self, objet_a_comparer):`	Opérateur d'égalité (*equal*). Renvoie `True` si `self` et `objet_a_comparer` sont égaux, `False` sinon.
`!=`	`def __ne__(self, objet_a_comparer):`	Différent de (*non equal*). Renvoie `True` si `self` et `objet_a_comparer` sont différents, `False` sinon.
`>`	`def __gt__(self, objet_a_comparer):`	Teste si `self` est strictement supérieur (*greather than*) à `objet_a_comparer`.
`>=`	`def __ge__(self, objet_a_comparer):`	Teste si `self` est supérieur ou égal (*greater or equal*) à `objet_a_comparer`.
`<`	`def __lt__(self, objet_a_comparer):`	Teste si `self` est strictement inférieur (*lower than*) à `objet_a_comparer`.
`<=`	`def __le__(self, objet_a_comparer):`	Teste si `self` est inférieur ou égal (*lower or equal*) à `objet_a_comparer`.

Sachez que ce sont ces méthodes spéciales qui sont appelées si, par exemple, vous voulez trier une liste contenant vos objets.

Sachez également que, si Python n'arrive pas à faire `objet1 < objet2`, il essayera l'opération inverse, soit `objet2 >= objet1`. Cela vaut aussi pour les autres opérateurs de comparaison que nous venons de voir.

Allez, je vais vous mettre deux exemples malgré tout, il ne tient qu'à vous de redéfinir les autres méthodes présentées plus haut :

```
def __eq__(self, autre_duree):
        """Test si self et autre_duree sont égales"""
        return self.sec == autre_duree.sec and self.min == autre_duree.min
    def __gt__(self, autre_duree):
        """Test si self > autre_duree"""
        # On calcule le nombre de secondes de self et autre_duree
        nb_sec1 = self.sec + self.min * 60
        nb_sec2 = autre_duree.sec + autre_duree.min * 60
        return nb_sec1 > nb_sec2
```

Ces exemples devraient vous suffire, je pense.

Des méthodes spéciales utiles à pickle

Vous vous souvenez de `pickle`, j'espère. Pour conclure ce chapitre sur les méthodes spéciales, nous allons en voir deux qui sont utilisées par ce module pour influencer la façon dont nos objets sont enregistrés dans des fichiers.

Prenons un cas concret, d'une utilité pratique discutable.

On crée une classe qui va contenir plusieurs attributs. Un de ces attributs possède une valeur temporaire, qui n'est utile que pendant l'exécution du programme. Si on arrête ce programme et qu'on le relance, on doit récupérer le même objet mais la valeur temporaire doit être remise à `0`, par exemple.

Il y a d'autres moyens d'y parvenir, je le reconnais. Mais les autres applications que j'ai en tête sont plus dures à développer et à expliquer rapidement, donc gardons cet exemple.

La méthode spéciale __getstate__

La méthode `__getstate__` est appelée au moment de sérialiser l'objet. Quand vous voulez enregistrer l'objet à l'aide du module `pickle`, `__getstate__` va être appelée juste avant l'enregistrement.

Si aucune méthode `__getstate__` n'est définie, `pickle` enregistre le dictionnaire des attributs de l'objet à enregistrer. Vous vous rappelez ? Il est contenu dans `objet.__dict__`.

Sinon, `pickle` enregistre dans le fichier la valeur renvoyée par `__getstate__` (généralement, un dictionnaire d'attributs modifié).

Voyons un peu comment coder notre exemple grâce à `__getstate__` :

```
class Temp:
    """Classe contenant plusieurs attributs, dont un temporaire"""
```

```

    def __init__(self):
        """Constructeur de notre objet"""
        self.attribut_1 = "une valeur"
        self.attribut_2 = "une autre valeur"
        self.attribut_temporaire = 5

    def __getstate__(self):
        """Renvoie le dictionnaire d'attributs à sérialiser"""
        dict_attr = dict(self.__dict__)
        dict_attr["attribut_temporaire"] = 0
        return dict_attr
```

Avant de revenir sur le code, vous pouvez en voir les effets. Si vous tentez d'enregistrer cet objet grâce à `pickle` et que vous le récupérez ensuite depuis le fichier, vous constatez que l'attribut `attribut_temporaire` est à `0`, peu importe sa valeur d'origine.

Voyons le code de `__getstate__`. La méthode ne prend aucun argument (excepté `self` puisque c'est une méthode d'instance).

Elle enregistre le dictionnaire des attributs dans une variable locale `dict_attr`. Ce dictionnaire a le même contenu que `self.__dict__` (le dictionnaire des attributs de l'objet). En revanche, il a une référence différente. Sans cela, à la ligne suivante, au moment de modifier `attribut_temporaire`, le changement aurait été également appliqué à l'objet, ce que l'on veut éviter.

À la ligne suivante, donc, on change la valeur de l'attribut `attribut_temporaire`. Étant donné que `dict_attr` et `self.__dict__` n'ont pas la même référence, l'attribut n'est changé que dans `dict_attr` et le dictionnaire de `self` n'est pas modifié.

Enfin, on renvoie `dict_attr`. Au lieu d'enregistrer dans notre fichier `self.__dict__`, `pickle` enregistre notre dictionnaire modifié, `dict_attr`.

Si ce n'est pas assez clair, je vous encourage à tester par vous-mêmes, essayez de modifier la méthode `__getstate__` et manipulez `self.__dict__` pour bien comprendre le code.

La méthode `__setstate__`

À la différence de `__getstate__`, la méthode `__setstate__` est appelée au moment de désérialiser l'objet. Concrètement, si vous récupérez un objet à partir d'un fichier sérialisé, `__setstate__` sera appelée après la récupération du dictionnaire des attributs.

Pour schématiser, voici l'exécution que l'on va observer derrière `unpickler.load()` :

1. L'objet **Unpickler** lit le fichier.
2. Il récupère le dictionnaire des attributs. Je vous rappelle que si aucune méthode `__getstate__` n'est définie dans notre classe, ce dictionnaire est celui contenu dans l'attribut spécial `__dict__` de l'objet au moment de sa sérialisation.
3. Ce dictionnaire récupéré est envoyé à la méthode `__setstate__` si elle existe. Si elle n'existe pas, Python considère que c'est le dictionnaire des attributs de

l'objet à récupérer et écrit donc l'attribut `__dict__` de l'objet en y plaçant ce dictionnaire récupéré.

Le même exemple mais, cette fois, par la méthode `__setstate__` :

```
...
    def \_\_setstate\_\_(self, dict\_attr):
        """Méthode appelée lors de la désérialisation de l'
            objet"""
        dict\_attr["attribut\_temporaire"] = 0
        self.\_\_dict\_\_ = dict\_attr
```



Quelle est la différence entre les deux méthodes que nous avons vues ?

L'objectif que nous nous étions fixé peut être atteint par ces deux méthodes. Soit notre classe met en œuvre une méthode `__getstate__`, soit elle met en œuvre une méthode `__setstate__`.

Dans le premier cas, on modifie le dictionnaire des attributs *avant* la sérialisation. Le dictionnaire des attributs enregistré est celui que nous avons modifié avec la valeur de notre attribut temporaire à `0`.

Dans le second cas, on modifie le dictionnaire d'attributs *après* la désérialisation. Le dictionnaire que l'on récupère contient un attribut `attribut_temporaire` avec une valeur quelconque (on ne sait pas laquelle) mais avant de récupérer l'objet, on met cette valeur à `0`.

Ce sont deux moyens différents, qui ici reviennent au même. À vous de choisir la meilleure méthode en fonction de vos besoins (les deux peuvent être présentes dans la même classe si nécessaire).

Là encore, je vous encourage à faire des essais si ce n'est pas très clair.

On peut enregistrer dans un fichier autre chose que des dictionnaires

Votre méthode `__getstate__` n'est pas obligée de renvoyer un dictionnaire d'attributs. Elle peut renvoyer un autre objet, un entier, un flottant, mais dans ce cas une méthode `__setstate__` devra exister pour savoir « quoi faire » avec l'objet enregistré. Si ce n'est pas un dictionnaire d'attributs, Python ne peut pas le deviner !

Là encore, je vous laisse tester si cela vous intéresse.

Je veux encore plus puissant !

`__getstate__` et `__setstate__` sont les deux méthodes les plus connues pour agir sur la sérialisation d'objets. Mais il en existe d'autres, plus complexes.

Si vous êtes intéressés, jetez un œil du côté de la PEP 307 *via* le code web suivant :

▷ PEP 307
Code web : 665199

En résumé

- Les méthodes spéciales permettent d'influencer la manière dont Python accède aux attributs d'une instance et réagit à certains opérateurs ou conversions.
- Les méthodes spéciales sont toutes entourées de deux signes « souligné » (`_`).
- Les méthodes `__getattr__`, `__setattr__` et `__delattr__` contrôlent l'accès aux attributs de l'instance.
- Les méthodes `__getitem__`, `__setitem__` et `__delitem__` surchargent l'indexation (`[]`).
- Les méthodes `__add__`, `__sub__`, `__mul__`... surchargent les opérateurs mathématiques.
- Les méthodes `__eq__`, `__ne__`, `__gt__`... surchargent les opérateurs de comparaison.

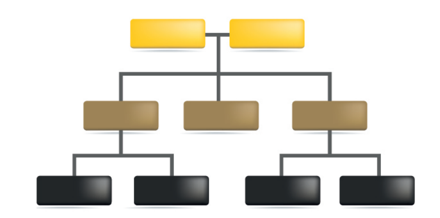
Chapitre 20

L'héritage

Difficulté : 

J'entends souvent dire qu'un langage de programmation orienté objet n'incluant pas l'héritage serait incomplet, sinon inutile. Après avoir découvert par moi-même cette fonctionnalité et les techniques qui en découlent, je suis forcé de reconnaître que sans l'héritage, le monde serait moins beau !

Qu'est-ce que cette fonctionnalité a de si utile ? Nous allons le voir, bien entendu. Et je vais surtout essayer de vous montrer des exemples d'applications. Car très souvent, quand on découvre l'héritage, on ne sait pas trop quoi en faire... Ne vous attendez donc pas à un chapitre où vous n'allez faire que coder. Vous allez devoir vous pencher sur de la théorie et travailler sur quelques exemples de modélisation. Mais je vous guide, ne vous inquiétez pas !



Pour bien commencer

Je ne vais pas faire durer le suspense plus longtemps : l'héritage est une fonctionnalité objet qui permet de déclarer que telle classe sera elle-même modelée sur une autre classe, qu'on appelle la classe parente, ou la **classe mère**. Concrètement, si une classe `b` **hérite** de la classe `a`, les objets créés sur le modèle de la classe `b` auront accès aux méthodes et attributs de la classe `a`.



Et c'est tout ? Cela ne sert à rien !

Non, ce n'est pas tout, et si, cela sert énormément mais vous allez devoir me laisser un peu de temps pour vous en montrer l'intérêt.

La première chose, c'est que la classe `b` dans notre exemple ne se contente pas de reprendre les méthodes et attributs de la classe `a` : elle va pouvoir en définir d'autres. D'autres méthodes et d'autres attributs qui lui seront propres, en plus des méthodes et attributs de la classe `a`. Et elle va pouvoir également redéfinir les méthodes de la classe mère.

Prenons un exemple simple : on a une classe `Animal` permettant de définir des animaux. Les animaux tels que nous les modélisons ont certains attributs (le régime : carnivore ou herbivore) et certaines méthodes (manger, boire, crier…).

On peut maintenant définir une classe `Chien` qui hérite de `Animal`, c'est-à-dire qu'elle reprend ses méthodes. Nous allons voir plus bas ce que cela implique exactement.

Si vous ne voyez pas très bien dans quel cas on fait hériter une classe d'une autre, faites le test :

- on fait hériter la classe `Chien` de `Animal` parce qu'*un chien est un animal* ;
- on ne fait pas hériter `Animal` de `Chien` parce qu'`Animal` n'est pas un `Chien`.

Sur ce modèle, vous pouvez vous rendre compte qu'*une voiture est un véhicule*. La classe `Voiture` pourrait donc hériter de `Vehicule`.

Intéressons-nous à présent au code.

L'héritage simple

On oppose l'**héritage simple**, dont nous venons de voir les aspects théoriques dans la section précédente, à l'**héritage multiple** que nous verrons dans la prochaine section.

Il est temps d'aborder la syntaxe de l'héritage. Nous allons définir une première classe `A` et une seconde classe `B` qui hérite de `A`.

```
class A:
    """Classe A, pour illustrer notre exemple d'héritage"""
    pass # On laisse la définition vide, ce n'est qu'un exemple

```

```
class B(A):
    """Classe B, qui hérite de A.
    Elle reprend les mêmes méthodes et attributs (dans cet
        exemple, la classe
    A ne possède de toute façon ni méthode ni attribut)"""

    pass
```

Vous pourrez expérimenter par la suite sur des exemples plus constructifs. Pour l'instant, l'important est de bien noter la syntaxe qui, comme vous le voyez, est des plus simples : `class MaClasse(MaClasseMere):`. Dans la définition de la classe, entre le nom et les deux points, vous précisez entre parenthèses la classe dont elle doit hériter. Comme je l'ai dit, dans un premier temps, toutes les méthodes de la classe `A` se retrouveront dans la classe `B`.



J'ai essayé de mettre des constructeurs dans les deux classes mais, dans la classe fille, je ne retrouve pas les attributs déclarés dans ma classe mère, c'est normal ?

Tout à fait. Vous vous souvenez quand je vous ai dit que les méthodes étaient définies dans la classe, alors que les attributs étaient directement déclarés dans l'instance d'objet ? Vous le voyez bien de toute façon : c'est dans le constructeur qu'on déclare les attributs et on les écrit tous dans l'instance `self`.

Quand une classe `B` hérite d'une classe `A`, les objets de type `B` reprennent bel et bien les méthodes de la classe `A` en même temps que celles de la classe `B`. Mais, assez logiquement, ce sont celles de la classe `B` qui sont appelées d'abord.

Si vous faites `objet_de_type_b.ma_methode()`, Python va d'abord chercher la méthode `ma_methode` dans la classe `B` dont l'objet est directement issu. S'il ne trouve pas, il va chercher récursivement dans les classes dont hérite `B`, c'est-à-dire `A` dans notre exemple. Ce mécanisme est très important : il induit que si aucune méthode n'a été redéfinie dans la classe, on cherche dans la classe mère. On peut ainsi redéfinir une certaine méthode dans une classe et laisser d'autres directement hériter de la classe mère.

Petit code d'exemple :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = "Martin"
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en
            chaîne"""
        return "{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):
```

```
    """Classe définissant un agent spécial.
    Elle hérite de la classe Personne"""

    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        self.nom = nom
        self.matricule = matricule
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""
        return "Agent {0}, matricule {1}".format(self.nom, self.matricule)
```

Vous voyez ici un exemple d'héritage simple. Seulement, si vous essayez de créer des agents spéciaux, vous risquez d'avoir de drôles de surprises :

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prenom
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'AgentSpecial' object has no attribute 'prenom'
>>>
```



Argh... mais tu n'avais pas dit qu'une classe reprenait les méthodes et attributs de sa classe mère ?

Si. Mais en suivant bien l'exécution, vous allez comprendre : tout commence à la création de l'objet. Quel constructeur appeler ? S'il n'y avait pas de constructeur défini dans notre classe `AgentSpecial`, Python appellerait celui de `Personne`. Mais il en existe bel et bien un dans la classe `AgentSpecial` et c'est donc celui-ci qui est appelé. Dans ce constructeur, on définit deux attributs, `nom` et `matricule`. Mais c'est tout : le constructeur de la classe `Personne` n'est pas appelé, sauf si vous l'appelez explicitement dans le constructeur d'`AgentSpecial`.

Dans le premier chapitre, je vous ai expliqué que `mon_objet.ma_methode()` revenait au même que `MaClasse.ma_methode(mon_objet)`. Dans notre méthode `ma_methode`, le premier paramètre `self` sera `mon_objet`. Nous allons nous servir de cette équivalence. La plupart du temps, écrire `mon_objet.ma_methode()` suffit. Mais dans une relation d'héritage, il peut y avoir, comme nous l'avons vu, plusieurs méthodes du même nom définies dans différentes classes. Laquelle appeler ? Python choisit, s'il la trouve, celle définie directement dans la classe dont est issu l'objet, et sinon parcourt la hiérarchie de l'héritage jusqu'à tomber sur la méthode. Mais on peut aussi se servir de la notation `MaClasse.ma_methode(mon_objet)` pour appeler une méthode précise d'une classe

précise. Et cela est utile dans notre cas :

```
class Personne:
    """Classe représentant une personne"""
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = "Martin"
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""
        return "{0} {1}".format(self.prenom, self.nom)

class AgentSpecial(Personne):
    """Classe définissant un agent spécial.
    Elle hérite de la classe Personne"""

    def __init__(self, nom, matricule):
        """Un agent se définit par son nom et son matricule"""
        # On appelle explicitement le constructeur de Personne :
        Personne.__init__(self, nom)
        self.matricule = matricule
    def __str__(self):
        """Méthode appelée lors d'une conversion de l'objet en chaîne"""
        return "Agent {0}, matricule {1}".format(self.nom, self.matricule)
```

Si cela vous paraît encore un peu vague, expérimentez : c'est toujours le meilleur moyen. Entraînez-vous, contrôlez l'écriture des attributs, ou revenez au premier chapitre de cette partie pour vous rafraîchir la mémoire au sujet du paramètre `self`, bien qu'à force de manipulations vous avez dû comprendre l'idée.

Reprenons notre code de tout à l'heure qui, cette fois, passe sans problème :

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> agent.nom
'Fisher'
>>> print(agent)
Agent Fisher, matricule 18327-121
>>> agent.prenom
'Martin'
>>>
```

Cette fois, notre attribut `prenom` se trouve bien dans notre agent spécial car le constructeur de la classe `AgentSpecial` appelle explicitement celui de `Personne`.

Vous pouvez noter également que, dans le constructeur d'`AgentSpecial`, on n'instancie pas l'attribut `nom`. Celui-ci est en effet écrit par le constructeur de la classe `Personne` que nous appelons en lui passant en paramètre le nom de notre agent.

Notez que l'on pourrait très bien faire hériter une nouvelle classe de notre classe `Personne`, la classe mère est souvent un modèle pour plusieurs classes filles.

Petite précision

Dans le chapitre précédent, je suis passé très rapidement sur l'héritage, ne voulant pas trop m'y attarder et brouiller les cartes inutilement. Mais j'ai expliqué brièvement que toutes les classes que vous créez héritent de la classe `object`. C'est elle, notamment, qui définit toutes les méthodes spéciales que nous avons vues au chapitre précédent et qui connaît, bien mieux que nous, le mécanisme interne de l'objet. Vous devriez un peu mieux, à présent, comprendre le code du chapitre précédent. Le voici, en substance :

```
def __setattr__(self, nom_attribut, valeur_attribut):
        """Méthode appelée quand on fait objet.attribut = valeur"""
        print("Attention, on modifie l'attribut {0} de l'objet !".format(nom_attribut))
        object.__setattr__(self, nom_attribut, valeur_attribut)
```

En redéfinissant la méthode `__setattr__`, on ne peut, dans le corps de cette méthode, modifier les valeurs de nos attributs comme on le fait habituellement (`self.attribut = valeur`) car alors, la méthode s'appellerait elle-même. On fait donc appel à la méthode `__setattr__` de la classe `object`, cette classe dont héritent implicitement toutes nos classes. On est sûr que la méthode de cette classe sait écrire une valeur dans un attribut, alors que nous ignorons le mécanisme et que nous n'avons pas besoin de le connaître : c'est la magie du procédé, une fois qu'on a bien compris le principe !

Deux fonctions très pratiques

Python définit deux fonctions qui peuvent se révéler utiles dans bien des cas : `issubclass` et `isinstance`.

issubclass

Comme son nom l'indique, elle vérifie si une classe est une sous-classe d'une autre classe. Elle renvoie `True` si c'est le cas, `False` sinon :

```
>>> issubclass(AgentSpecial, Personne) # AgentSpecial hérite de Personne
True
>>> issubclass(AgentSpecial, object)
True
>>> issubclass(Personne, object)
True
>>> issubclass(Personne, AgentSpecial) # Personne n'hérite pas d'AgentSpecial
False
```

```
>>>
```

`isinstance`

`isinstance` permet de savoir si un objet est issu d'une classe ou de ses classes filles :

```
>>> agent = AgentSpecial("Fisher", "18327-121")
>>> isinstance(agent, AgentSpecial) # Agent est une instance d'
    AgentSpecial
True
>>> isinstance(agent, Personne) # Agent est une instance hérité
    e de Personne
True
>>>
```

Ces quelques exemples suffisent, je pense. Peut-être devrez-vous attendre un peu avant de trouver une utilité à ces deux fonctions mais ce moment viendra.

L'héritage multiple

Python inclut un mécanisme permettant l'**héritage multiple**. L'idée est en substance très simple : au lieu d'hériter d'une seule classe, on peut hériter de plusieurs.



Ce n'est pas ce qui se passe quand on hérite d'une classe qui hérite elle-même d'une autre classe ?

Pas tout à fait. La hiérarchie de l'héritage simple permet d'étendre des méthodes et attributs d'une classe à plusieurs autres, mais la structure reste fermée. Pour mieux comprendre, considérez l'exemple qui suit.

On peut s'asseoir dans un fauteuil. On peut dormir dans un lit. Mais on peut s'asseoir et dormir dans certains canapés (la plupart en fait, avec un peu de bonne volonté). Notre classe `Fauteuil` pourra hériter de la classe `ObjetPourSAsseoir` et notre classe `Lit`, de notre classe `ObjetPourDormir`. Mais notre classe `Canape` alors ? Elle devra logiquement hériter de nos deux classes `ObjetPourSAsseoir` et `ObjetPourDormir`. C'est un cas où l'héritage multiple pourrait se révéler utile.

Assez souvent, on utilisera l'héritage multiple pour des classes qui ont besoin de certaines fonctionnalités définies dans une classe mère. Par exemple, une classe peut produire des objets destinés à être enregistrés dans des fichiers. On peut faire hériter de cette classe toutes celles qui produiront des objets à enregistrer dans des fichiers. Mais ces mêmes classes pourront hériter d'autres classes incluant, pourquoi pas, d'autres fonctionnalités.

C'est une des utilisations de l'héritage multiple et il en existe d'autres. Bien souvent, l'utilisation de cette fonctionnalité ne vous semblera évidente qu'en vous penchant sur

la hiérarchie d'héritage de votre programme. Pour l'instant, je vais me contenter de vous donner la syntaxe et un peu de théorie supplémentaire, en vous encourageant à essayer par vous-mêmes :

```
class MaClasseHeritee(MaClasseMere1, MaClasseMere2):
```

Vous pouvez faire hériter votre classe de plus de deux autres classes. Au lieu de préciser, comme dans les cas d'héritage simple, une seule classe mère entre parenthèses, vous en indiquez plusieurs, séparées par des virgules.

Recherche des méthodes

La recherche des méthodes se fait dans l'ordre de la définition de la classe. Dans l'exemple ci-dessus, si on appelle une méthode d'un objet issu de `MaClasseHeritee`, on va d'abord chercher dans la classe `MaClasseHeritee`. Si la méthode n'est pas trouvée, on la cherche d'abord dans `MaClasseMere1`. Encore une fois, si la méthode n'est pas trouvée, on cherche dans toutes les classes mères de la classe `MaClasseMere1`, si elle en a, et selon le même système. Si, encore et toujours, on ne trouve pas la méthode, on la recherche dans `MaClasseMere2` et ses classes mères successives.

C'est donc l'ordre de définition des classes mères qui importe. On va chercher la méthode dans les classes mères de gauche à droite. Si on ne trouve pas la méthode dans une classe mère donnée, on remonte dans ses classes mères, et ainsi de suite.

Retour sur les exceptions

Depuis la première partie, nous ne sommes pas revenus sur les exceptions. Toutefois, ce chapitre me donne une opportunité d'aller un peu plus loin.

Les exceptions sont non seulement des classes, mais des classes hiérarchisées selon une relation d'héritage précise.

Cette relation d'héritage devient importante quand vous utilisez le mot-clé `except`. En effet, le type de l'exception que vous précisez après est intercepté... ainsi que toutes les classes qui héritent de ce type.



Mais comment fait-on pour savoir qu'une exception hérite d'autres exceptions ?

Il y a plusieurs possibilités. Si vous vous intéressez à une exception en particulier, consultez l'aide qui lui est liée.

```
Help on class AttributeError in module builtins:

class AttributeError(Exception)
 |  Attribute not found.
```

```
 |
 |   Method resolution order:
 |       AttributeError
 |       Exception
 |       BaseException
 |       object
```

Vous apprenez ici que l'exception `AttributeError` hérite de `Exception`, qui hérite elle-même de `BaseException`.

Vous pouvez également retrouver la hiérarchie des exceptions *built-in* sur le site de Python, via ce code web :

▷ Hiérarchie des exceptions
Code web : 360446

Ne sont répertoriées ici que les exceptions dites *built-in*. D'autres peuvent être définies dans des modules que vous utiliserez et vous pouvez même en créer vous-mêmes (nous allons voir cela un peu plus bas).

Pour l'instant, souvenez-vous que, quand vous écrivez `except TypeException`, vous pourrez intercepter toutes les exceptions du type `TypeException` mais aussi celles des classes héritées de `TypeException`.

La plupart des exceptions sont levées pour signaler une erreur... mais pas toutes. L'exception `KeyboardInterupt` est levée quand vous interrompez votre programme, par exemple avec CTRL + C. Si bien que, quand on souhaite intercepter toutes les erreurs potentielles, on évitera d'écrire un simple `except:` et on le remplacera par `except Exception:`, toutes les exceptions « d'erreurs » étant dérivées de `Exception`.

Création d'exceptions personnalisées

Il peut vous être utile de créer vos propres exceptions. Puisque les exceptions sont des classes, comme nous venons de le voir, rien ne vous empêche de créer les vôtres. Vous pourrez les lever avec `raise`, les intercepter avec `except`.

Se positionner dans la hiérarchie

Vos exceptions doivent hériter d'une exception *built-in* proposée par Python. Commencez par parcourir la hiérarchie des exceptions *built-in* pour voir si votre exception peut être dérivée d'une exception qui lui serait proche. La plupart du temps, vous devrez choisir entre ces deux exceptions :

- `BaseException` : la classe mère de *toutes* les exceptions. La plupart du temps, si vous faites hériter votre classe de `BaseException`, ce sera pour modéliser une exception qui ne sera pas foncièrement une erreur, par exemple une interruption dans le traitement de votre programme.
- `Exception` : c'est de cette classe que vos exceptions hériteront la plupart du temps. C'est la classe mère de toutes les exceptions « d'erreurs ».

Si vous pouvez trouver, dans le contexte, une exception qui se trouve plus bas dans la hiérarchie, c'est toujours mieux.



Que doit contenir notre classe exception ?

Deux choses : un constructeur et une méthode `__str__` car, au moment où l'exception est levée, elle doit être affichée. Souvent, votre constructeur ne prend en paramètre que le message d'erreur et la méthode `__str__` renvoie ce message :

```python
class MonException(Exception):
    """Exception levée dans un certain contexte..qui reste à dé
        finir"""
    def __init__(self, message):
        """On se contente de stocker le message d'erreur"""
        self.message = message
    def __str__(self):
        """On renvoie le message"""
        return self.message
```

Cette exception s'utilise le plus simplement du monde :

```
>>> raise MonException("OUPS... j'ai tout cassé")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MonException: OUPS... j'ai tout cassé
>>>
```

Mais vos exceptions peuvent aussi prendre plusieurs paramètres à l'instanciation :

```python
class ErreurAnalyseFichier(Exception):
    """Cette exception est levée quand un fichier (de
        configuration)
    n'a pas pu être analysé.

    Attributs :
        fichier -- le nom du fichier posant problème
        ligne -- le numéro de la ligne posant problème
        message -- le problème proprement dit"""

    def __init__(self, fichier, ligne, message):
        """Constructeur de notre exception"""
        self.fichier = fichier
        self.ligne = ligne
        self.message = message
    def __str__(self):
        """Affichage de l'exception"""
        return "[{}:{}]: {}".format(self.fichier, self.ligne, \
                self.message)
```

Et pour lever cette exception :

```
>>> raise ErreurAnalyseFichier("plop.conf", 34,
...         "Il manque une parenthèse à la fin de l'expression")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.ErreurAnalyseFichier: [plop.conf:34]: il manque une parenthèse à la fin de l'expression
>>>
```

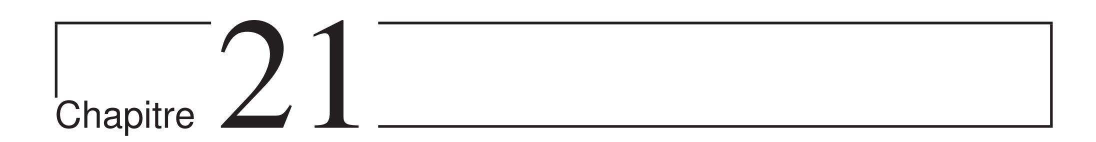
Voilà, ce petit retour sur les exceptions est achevé. Si vous voulez en savoir plus, n'hésitez pas à consulter la documentation Python les concernant, accessible grâce aux codes web suivants :

▷ Les exceptions
Code web : 777269

▷ Les exceptions personnalisées
Code web : 114349

En résumé

– L'héritage permet à une classe d'hériter du comportement d'une autre en reprenant ses méthodes.
– La syntaxe de l'héritage est `class NouvelleClasse(ClasseMere):`.
– On peut accéder aux méthodes de la classe mère directement via la syntaxe : `ClasseMere.methode(self)`.
– L'héritage multiple permet à une classe d'hériter de plusieurs classes mères.
– La syntaxe de l'héritage multiple s'écrit donc de la manière suivante : `class NouvelleClasse(ClasseMere1, ClasseMere2, ClasseMereN):`.
– Les exceptions définies par Python sont ordonnées selon une hiérarchie d'héritage.



Chapitre 21

Derrière la boucle for

Difficulté : 

Voilà pas mal de chapitres, nous avons étudié les boucles. Ne vous alarmez pas, ce que nous avons vu est toujours d'actualité ... mais nous allons un peu approfondir le sujet, maintenant que nous explorons le monde de l'objet.

Nous allons ici parler d'**itérateurs** et de **générateurs**. Nous allons découvrir ces concepts du plus simple au plus complexe et de telle sorte que chacun des concepts abordés reprenne les précédents. N'hésitez pas, par la suite, à revenir sur ce chapitre et à le relire, partiellement ou intégralement si nécessaire.



Les itérateurs

Nous utilisons des itérateurs sans le savoir depuis le moment où nous avons abordé les boucles et surtout, depuis que nous utilisons le mot-clé `for` pour parcourir des objets conteneurs.

```
ma_liste = [1, 2, 3]
for element in ma_liste:
```

Utiliser les itérateurs

C'est sur la seconde ligne que nous allons nous attarder : à force d'utiliser ce type de syntaxe, vous avez dû vous y habituer et ce type de parcours doit vous être familier. Mais il se cache bel et bien un mécanisme derrière cette instruction.

Quand Python tombe sur une ligne du type `for element in ma_liste:`, il va appeler l'itérateur de `ma_liste`. L'itérateur, c'est un objet qui va être chargé de parcourir l'objet conteneur, ici une liste.

L'itérateur est créé dans la méthode spéciale `__iter__` de l'objet. Ici, c'est donc la méthode `__iter__` de la classe `list` qui est appelée et qui renvoie un itérateur permettant de parcourir la liste.

À chaque tour de boucle, Python appelle la méthode spéciale `__next__` de l'itérateur, qui doit renvoyer l'élément suivant du parcours ou lever l'exception `StopIteration` si le parcours touche à sa fin.

Ce n'est peut-être pas très clair… alors voyons un exemple.

Avant de plonger dans le code, sachez que Python utilise deux fonctions pour appeler et manipuler les itérateurs : `iter` permet d'appeler la méthode spéciale `__iter__` de l'objet passé en paramètre et `next` appelle la méthode spéciale `__next__` de l'itérateur passé en paramètre.

```
>>> ma_chaine = "test"
>>> iterateur_de_ma_chaine = iter(ma_chaine)
>>> iterateur_de_ma_chaine
<str_iterator object at 0x00B408F0>
>>> next(iterateur_de_ma_chaine)
't'
>>> next(iterateur_de_ma_chaine)
'e'
>>> next(iterateur_de_ma_chaine)
's'
>>> next(iterateur_de_ma_chaine)
't'
>>> next(iterateur_de_ma_chaine)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>>
```

- On commence par créer une chaîne de caractères (jusque là, rien de compliqué).
- On appelle ensuite la fonction `iter` en lui passant en paramètre la chaîne. Cette fonction appelle la méthode spéciale `__iter__` de la chaîne, qui renvoie l'itérateur permettant de parcourir `ma_chaine`.
- On va ensuite appeler plusieurs fois la fonction `next` en lui passant en paramètre l'itérateur. Cette fonction appelle la méthode spéciale `__next__` de l'itérateur. Elle renvoie successivement chaque lettre contenue dans notre chaîne et lève une exception `StopIteration` quand la chaîne a été parcourue entièrement.

Quand on parcourt une chaîne grâce à une boucle `for` (`for lettre in chaine:`), c'est ce mécanisme d'itérateur qui est appelé. Chaque lettre renvoyée par notre itérateur se retrouve dans la variable `lettre` et la boucle s'arrête quand l'exception `StopIteration` est levée.

Vous pouvez reprendre ce code avec d'autres objets conteneurs, des listes par exemple.

Créons nos itérateurs

Pour notre exemple, nous allons créer deux classes :

- `RevStr` : une classe héritant de `str` qui se contentera de redéfinir la méthode `__iter__`. Son mode de parcours sera ainsi altéré : au lieu de parcourir la chaîne de gauche à droite, on la parcourra de droite à gauche (de la dernière lettre à la première).
- `ItRevStr` : notre itérateur. Il sera créé depuis la méthode `__iter__` de `RevStr` et devra parcourir notre chaîne du dernier caractère au premier.

Ce mécanisme est un peu nouveau, je vous mets le code sans trop de suspense. Si vous vous sentez de faire l'exercice, n'hésitez pas, mais je vous donnerai de toute façon l'occasion de pratiquer dès le prochain chapitre.

```
class RevStr(str):
    """Classe reprenant les méthodes et attributs des chaînes construites
    depuis 'str'. On se contente de définir une méthode de parcours
    différente : au lieu de parcourir la chaîne de la première à la dernière
    lettre, on la parcourt de la dernière à la première.

    Les autres méthodes, y compris le constructeur, n'ont pas besoin
    d'être redéfinies"""

    def __iter__(self):
        """Cette méthode renvoie un itérateur parcourant la chaîne
        dans le sens inverse de celui de 'str'"""

```

```python
        return ItRevStr(self) # On renvoie l'itérateur créé pour l'occasion

class ItRevStr:
    """Un itérateur permettant de parcourir une chaîne de la dernière lettre
    à la première. On stocke dans des attributs la position courante et la
    chaîne à parcourir"""

    def __init__(self, chaine_a_parcourir):
       """On se positionne à la fin de la chaîne"""
        self.chaine_a_parcourir = chaine_a_parcourir
        self.position = len(chaine_a_parcourir)
    def __next__(self):
        """Cette méthode doit renvoyer l'élément suivant dans le parcours,
        ou lever l'exception 'StopIteration' si le parcours est fini"""

        if self.position == 0: # Fin du parcours
            raise StopIteration
        self.position -= 1 # On décrémente la position
        return self.chaine_a_parcourir[self.position]
```

À présent, vous pouvez créer des chaînes devant se parcourir du dernier caractère vers le premier.

```python
>>> ma_chaine = RevStr("Bonjour")
>>> ma_chaine
'Bonjour'
>>> for lettre in ma_chaine:
...     print(lettre)
...
r
u
o
j
n
o
B
>>>
```

Sachez qu'il est aussi possible de mettre en œuvre directement la méthode `__next__` dans notre objet conteneur. Dans ce cas, la méthode `__iter__` pourra renvoyer `self`. Vous pouvez voir un exemple, dont le code ci-dessus est inspiré, en utilisant le code web suivant :

▷ Exemple sur les itérateurs
Code web : 547173



Cela reste quand même plutôt lourd non, de devoir faire des itérateurs à chaque fois ? Surtout si nos objets conteneurs doivent se parcourir de plusieurs façons, comme les dictionnaires par exemple.

Oui, il subsiste quand même beaucoup de répétitions dans le code que nous devons produire, surtout si nous devons créer plusieurs itérateurs pour un même objet. Souvent, on utilisera des itérateurs existants, par exemple celui des listes. Mais il existe aussi un autre mécanisme, plus simple et plus intuitif : la raison pour laquelle je ne vous montre pas en premier cette autre façon de faire, c'est que cette autre façon passe quand même par des itérateurs, même si c'est implicite, et qu'il n'est pas mauvais de savoir comment cela marche en coulisse.

Il est temps à présent de jeter un coup d'œil du côté des générateurs.

Les générateurs

Les générateurs sont avant tout un moyen plus pratique de créer et manipuler des itérateurs. Vous verrez un peu plus loin dans ce chapitre qu'ils permettent des choses assez complexes, mais leur puissance tient surtout à leur simplicité et leur petite taille.

Les générateurs simples

Pour créer des générateurs, nous allons découvrir un nouveau mot-clé : `yield`. Ce mot-clé ne peut s'utiliser que dans le corps d'une fonction et il est suivi d'une valeur à renvoyer.



Attends un peu... une valeur ? À renvoyer ?

Oui. Le principe des générateurs étant un peu particulier, il nécessite un mot-clé pour lui tout seul. L'idée consiste à définir une fonction pour un type de parcours. Quand on demande le premier élément du parcours (grâce à `next`), la fonction commence son exécution. Dès qu'elle rencontre une instruction `yield`, elle renvoie la valeur qui suit et se met en pause. Quand on demande l'élément suivant de l'objet (grâce, une nouvelle fois, à `next`), l'exécution reprend à l'endroit où elle s'était arrêtée et s'interrompt au `yield` suivant... et ainsi de suite. À la fin de l'exécution de la fonction, l'exception `StopIteration` est automatiquement levée par Python.

Nous allons prendre un exemple très simple pour commencer :

```
>>> def mon_generateur():
...     """Notre premier générateur. Il va simplement renvoyer
    1, 2 et 3"""
...     yield 1
...     yield 2
```

```
...         yield 3
...
>>> mon_generateur
<function mon_generateur at 0x00B494F8>
>>> mon_generateur()
<generator object mon_generateur at 0x00B9DC88>
>>> mon_iterateur = iter(mon_generateur())
>>> next(mon_iterateur)
1
>>> next(mon_iterateur)
2
>>> next(mon_iterateur)
3
>>> next(mon_iterateur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Je pense que cela vous rappelle quelque chose ! Cette fonction, à part l'utilisation de `yield`, est plutôt classique. Quand on l'exécute, on se retrouve avec un générateur. Ce générateur est un objet créé par Python qui définit sa propre méthode spéciale `__iter__` et donc son propre itérateur. Nous aurions tout aussi bien pu faire :

```
for nombre in mon_generateur(): # Attention on exécute la
    fonction
    print(nombre)
```

Cela rend quand même le code bien plus simple à comprendre.

Notez qu'on doit exécuter la fonction `mon_generateur` pour obtenir un générateur. Si vous essayez de parcourir notre fonction (`for nombre in mon_generateur`), cela ne marchera pas.

Bien entendu, la plupart du temps, on ne se contentera pas d'appeler `yield` comme cela. Le générateur de notre exemple n'a pas beaucoup d'intérêt, il faut bien le reconnaître.

Essayons de faire une chose un peu plus utile : un générateur prenant en paramètres deux entiers, une borne inférieure et une borne supérieure, et renvoyant chaque entier compris entre ces bornes. Si on écrit par exemple `intervalle(5, 10)`, on pourra parcourir les entiers de 6 à 9.

Le résultat attendu est donc :

```
>>> for nombre in intervalle(5, 10):
...     print(nombre)
...
6
7
8
9
>>>
```

Vous pouvez essayer de faire l'exercice, c'est un bon entraînement et pas très compliqué de surcroît.

Au cas où, voici la correction :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre
        borne_inf et borne_sup.

    Note: borne_inf doit être inférieure à borne_sup"""

    borne_inf += 1
    while borne_inf < borne_sup:
        yield borne_inf
        borne_inf += 1
```

Là encore, vous pouvez améliorer cette fonction. Pourquoi ne pas faire en sorte que, si la borne inférieure est supérieure à la borne supérieure, le parcours se fasse dans l'autre sens ?

L'important est que vous compreniez bien l'intérêt et le mécanisme derrière. Je vous encourage, là encore, à tester, à disséquer cette fonctionnalité, à essayer de reprendre les exemples d'itérateurs et à les convertir en générateurs.

Si, dans une classe quelconque, la méthode spéciale `__iter__` contient un appel à `yield`, alors ce sera ce générateur qui sera appelé quand on voudra parcourir la boucle. Même quand Python passe par des générateurs, comme vous l'avez vu, il utilise (implicitement) des itérateurs. C'est juste plus confortable pour le codeur, on n'a pas besoin de créer une classe par itérateur ni de coder une méthode `__next__`, ni même de lever l'exception `StopIteration` : Python fait tout cela pour nous. Pratique non ?

Les générateurs comme co-routines

Jusqu'ici, que ce soit avec les itérateurs ou avec les générateurs, nous créons un moyen de parcourir notre objet au début de la boucle `for`, en sachant que nous ne pourrons pas modifier le comportement du parcours par la suite. Mais les générateurs possèdent un certain nombre de méthodes permettant, justement, d'interagir avec eux pendant le parcours.

Malheureusement, à notre niveau, les idées d'applications *utiles* me manquent et je vais me contenter de vous présenter la syntaxe et un petit exemple. Peut-être trouverez-vous par la suite une application utile des **co-routines** quand vous vous lancerez dans des programmes conséquents, ou que vous aurez été plus loin dans l'apprentissage du Python.

Les **co-routines** sont un moyen d'altérer le parcours... pendant le parcours. Par exemple, dans notre générateur `intervalle`, on pourrait vouloir passer directement de 5 à 10.

Le système des co-routines en Python est contenu dans le mot-clé `yield` que nous avons vu plus haut et l'utilisation de certaines méthodes de notre générateur.

Interrompre la boucle

La première méthode que nous allons voir est `close`. Elle permet d'interrompre prématurément la boucle, comme le mot-clé `break` en somme.

```
generateur = intervalle(5, 20)
for nombre in generateur:
    if nombre > 17:
        generateur.close() # Interruption de la boucle
```

Comme vous le voyez, pour appeler les méthodes du générateur, on doit le stocker dans une variable avant la boucle. Si vous aviez écrit directement `for nombre in intervalle(5, 20)`, vous n'auriez pas pu appeler la méthode `close` du générateur.

Envoyer des données à notre générateur

Pour cet exemple, nous allons étendre notre générateur pour qu'il accepte de recevoir des données pendant son exécution.

Le point d'échange de données se fait au mot-clé `yield`. `yield valeur` « renvoie » `valeur` qui deviendra donc la valeur courante du parcours. La fonction se met ensuite en pause. On peut, à cet instant, envoyer une valeur à notre générateur. Cela permet d'altérer le fonctionnement de notre générateur pendant le parcours.

Reprenons notre exemple en intégrant cette fonctionnalité :

```
def intervalle(borne_inf, borne_sup):
    """Générateur parcourant la série des entiers entre borne_inf et borne_sup.
    Notre générateur doit pouvoir "sauter" une certaine plage de nombres
    en fonction d'une valeur qu'on lui donne pendant le parcours. La
    valeur qu'on lui passe est la nouvelle valeur de borne_inf.

    Note: borne_inf doit être inférieure à borne_sup"""
    borne_inf += 1
    while borne_inf < borne_sup:
        valeur_recue = (yield borne_inf)
        if valeur_recue is not None: # Notre générateur a reçu quelque chose
            borne_inf = valeur_recue
        borne_inf += 1
```

Nous configurons notre générateur pour qu'il accepte une valeur éventuelle au cours du parcours. S'il reçoit une valeur, il va l'attribuer au point du parcours.

Autrement dit, au cours de la boucle, vous pouvez demander au générateur de sauter tout de suite à 20 si le nombre est 15.

Tout se passe à partir de la ligne du `yield`. Au lieu de simplement renvoyer une valeur à notre boucle, on capture une éventuelle valeur dans `valeur_recue`. La syntaxe est

simple : `variable = (yield valeur_a_renvoyer)` [1].

Si aucune valeur n'a été passée à notre générateur, notre `valeur_recue` vaudra `None`. On vérifie donc si elle ne vaut pas `None` et, dans ce cas, on affecte la nouvelle valeur à `borne_inf`.

Voici le code permettant d'interagir avec notre générateur. On utilise la méthode `send` pour envoyer une valeur à notre générateur :

```
generateur = intervalle(10, 25)
for nombre in generateur:
    if nombre == 15: # On saute à 20
        generateur.send(20)
    print(nombre, end=" ")
```

Il existe d'autres méthodes permettant d'interagir avec notre générateur. Vous pouvez les retrouver, ainsi que des explications supplémentaires, sur la documentation officielle traitant du mot-clé `yield` avec le code web suivant :

▷ Le mot-clé yield
Code web : 302240

En résumé

- Quand on utilise la boucle `for element in sequence:`, un itérateur de cette séquence permet de la parcourir.
- On peut récupérer l'itérateur d'une séquence grâce à la fonction `iter`.
- Une séquence renvoie l'itérateur permettant de la parcourir grâce à la méthode spéciale `__iter__`.
- Un itérateur possède une méthode spéciale, `__next__`, qui renvoie le prochain élément à parcourir ou lève l'exception `StopIteration` qui arrête la boucle.
- Les générateurs permettent de créer plus simplement des itérateurs.
- Ce sont des fonctions utilisant le mot-clé `yield` suivi de la valeur à transmettre à la boucle.

1. N'oubliez pas les parenthèses autour de `yield valeur`.

Chapitre 22

TP : un dictionnaire ordonné

Difficulté : 

Voici enfin le moment de la pratique. Vous avez appris pas mal de choses dans cette partie, beaucoup de concepts, souvent théoriques. Il est temps de les mettre en application, dans un contexte un peu différent des TP précédents : on ne va pas créer un jeu mais plutôt un objet conteneur tenant à la fois du dictionnaire et de la liste.



Notre mission

Notre énoncé va être un peu différent de ceux dont vous avez l'habitude. Nous n'allons pas créer ici un jeu mais simplement une classe, destinée à produire des objets conteneurs, des dictionnaires ordonnés.

Peut-être ne vous en souvenez-vous pas mais je vous ai dit dans le chapitre consacré aux dictionnaires que c'était un type non-ordonné. Ainsi, l'ordre dans lequel vous entrez les données n'a pas d'importance. On ne peut ni les trier, ni les inverser, tout cela n'aurait aucun sens pour ce type particulier.

Mais nous allons profiter de l'occasion pour créer une forme de dictionnaire ordonné. L'idée, assez simplement, est de stocker nos données dans deux listes :

- la première contenant nos clés ;
- la seconde contenant les valeurs correspondantes.

L'ordre d'ajout sera ainsi important, on pourra trier et inverser ce type de dictionnaire.

Spécifications

Voici la liste des mécanismes que notre classe devra mettre en œuvre. Un peu plus bas, vous trouverez un exemple de manipulation de l'objet qui reprend ces spécifications :

1. On doit pouvoir créer le dictionnaire de plusieurs façons :
   - Vide : on appelle le constructeur sans lui passer aucun paramètre et le dictionnaire créé est donc vide.
   - Copié depuis un dictionnaire : on passe en paramètre du constructeur un dictionnaire que l'on copie par la suite dans notre objet. On peut ainsi écrire `constructeur(dictionnaire)` et les clés et valeurs contenues dans le dictionnaire sont copiées dans l'objet construit.
   - Pré-rempli grâce à des clés et valeurs passées en paramètre : comme les dictionnaires usuels, on doit ici avoir la possibilité de pré-remplir notre objet avec des couples clés-valeurs passés en paramètre (`constructeur(cle1 = valeur1, cle2 = valeur2, ...)`).
2. Les clés et valeurs doivent être couplées. Autrement dit, si on cherche à supprimer une clé, la valeur correspondante doit également être supprimée. Les clés et valeurs se trouvant dans des listes de même taille, il suffira de prendre l'indice dans une liste pour savoir quel objet lui correspond dans l'autre. Par exemple, la clé d'indice `0` est couplée avec la valeur d'indice `0`.
3. On doit pouvoir interagir avec notre objet conteneur grâce aux crochets, pour récupérer une valeur (`objet[cle]`), pour la modifier (`objet[cle] = valeur`) ou pour la supprimer (`del objet[cle]`).
4. Quand on cherche à modifier une valeur, si la clé existe on écrase l'ancienne valeur, si elle n'existe pas on ajoute le couple clé-valeur à la fin du dictionnaire.
5. On doit pouvoir savoir grâce au mot-clé `in` si une clé se trouve dans notre dictionnaire (`cle in dictionnaire`).

6. On doit pouvoir demander la taille du dictionnaire grâce à la fonction `len`.
7. On doit pouvoir afficher notre dictionnaire directement dans l'interpréteur ou grâce à la fonction `print`. L'affichage doit être similaire à celui des dictionnaires usuels (`{cle1: valeur1, cle2: valeur2, ...}`).
8. L'objet doit définir les méthodes `sort` pour le trier et `reverse` pour l'inverser. Le tri de l'objet doit se faire en fonction des clés.
9. L'objet doit pouvoir être parcouru. Quand on écrit `for cle in dictionnaire`, on doit parcourir la liste des clés contenues dans le dictionnaire.
10. À l'instar des dictionnaires, trois méthodes `keys()` (renvoyant la liste des clés), `values()` (renvoyant la liste des valeurs) et `items()` (renvoyant les couples (clé, valeur)) doivent être mises en œuvre. Le type de retour de ces méthodes est laissé à votre initiative : il peut s'agir d'itérateurs ou de générateurs (tant qu'on peut les parcourir).
11. On doit pouvoir ajouter deux dictionnaires ordonnés (`dico1 + dico2`) ; les clés et valeurs du second dictionnaire sont ajoutées au premier.

Cela vous en fait, du boulot !

Et vous pourrez encore trouver le moyen d'améliorer votre classe par la suite, si vous le désirez.

Exemple de manipulation

Ci-dessous se trouve un exemple de manipulation de notre dictionnaire ordonné. Quand vous aurez codé le vôtre, vous pourrez voir s'il réagit de la même façon que le mien.

```
>>> fruits = DictionnaireOrdonne()
>>> fruits
{}
>>> fruits["pomme"] = 52
>>> fruits["poire"] = 34
>>> fruits["prune"] = 128
>>> fruits["melon"] = 15
>>> fruits
{'pomme': 52, 'poire': 34, 'prune': 128, 'melon': 15}
>>> fruits.sort()
>>> print(fruits)
{'melon': 15, 'poire': 34, 'pomme': 52, 'prune': 128}
>>> legumes = DictionnaireOrdonne(carotte = 26, haricot = 48)
>>> print(legumes)
{'carotte': 26, 'haricot': 48}
>>> len(legumes)
2
>>> legumes.reverse()
>>> fruits = fruits + legumes
>>> fruits
{'melon': 15, 'poire': 34, 'pomme': 52, 'prune': 128, 'haricot
    ': 48, 'carotte':
```

```
26}
>>> del fruits['haricot']
>>> 'haricot' in fruits
False
>>> legumes['haricot']
48
>>> for cle in legumes:
...     print(cle)
...
haricot
carotte
>>> legumes.keys()
['haricot', 'carotte']
>>> legumes.values()
[48, 26]
>>> for nom, qtt in legumes.items():
...     print("{0} ({1})".format(nom, qtt))
...
haricot (48)
carotte (26)
>>>
```

Tous au départ !

Je vous ai donné le nécessaire, c'est maintenant à vous de jouer. Concernant l'implémentation, les fonctionnalités, il reste des zones obscures, c'est volontaire. Tout ce qui n'est pas clairement dit est à votre initiative. Tant que cela fonctionne et que l'exemple de manipulation ci-dessus affiche la même chose chez vous, c'est parfait. Si vous voulez mettre en œuvre d'autres fonctionnalités, méthodes ou attributs, ne vous gênez pas... mais n'oubliez pas d'y aller progressivement.

C'est parti !

Correction proposée

Voici la correction que je vous propose. Je suis sûr que vous êtes, de votre côté, arrivés à quelque chose, même si tout ne fonctionne pas encore parfaitement. Certaines fonctionnalités, comme le tri, l'affichage, etc. sont encore un peu complexes à appréhender. Cependant, faites attention à ne pas sauter trop rapidement à la correction et essayez au moins d'obtenir par vous-mêmes un dictionnaire ordonné avec des fonctionnalités opérationnelles d'ajout, de consultation et de suppression d'éléments.

▷ Copier ce code
Code web : 167286



ATTENTION LES YEUX...

```python
class DictionnaireOrdonne:
    """Notre dictionnaire ordonné. L'ordre des données est maintenu
    et il peut donc, contrairement aux dictionnaires usuels, être trié
    ou voir l'ordre de ses données inversées"""

    def __init__(self, base={}, **donnees):
        """Constructeur de notre objet. Il peut ne prendre aucun paramètre
        (dans ce cas, le dictionnaire sera vide) ou construire un
        dictionnaire remplis grâce :
        -   au dictionnaire 'base' passé en premier paramètre ;
        -   aux valeurs que l'on retrouve dans 'donnees'."""

        self._cles = [] # Liste contenant nos clés
        self._valeurs = [] # Liste contenant les valeurs correspondant à nos clés

        # On vérifie que 'base' est un dictionnaire exploitable
        if type(base) not in (dict, DictionnaireOrdonne):
            raise TypeError( \
                "le type attendu est un dictionnaire (usuel ou ordonne)")

        # On récupère les données de 'base'
        for cle in base:
            self[cle] = base[cle]

        # On récupère les données de 'donnees'
        for cle in donnees:
            self[cle] = donnees[cle]

    def __repr__(self):
        """Représentation de notre objet. C'est cette chaîne qui sera affichée
        quand on saisit directement le dictionnaire dans l'interpréteur, ou en
        utilisant la fonction 'repr'"""

        chaine = "{"
        premier_passage = True
        for cle, valeur in self.items():
            if not premier_passage:
                chaine += ", " # On ajoute la virgule comme sé
```

```
parateur
            else:
                premier_passage = False
            chaine += repr(cle) + ": " + repr(valeur)
        chaine += "}"
        return chaine

    def __str__(self):
        """Fonction appelée quand on souhaite afficher le dictionnaire grâce
        à la fonction 'print' ou le convertir en chaîne grâce au constructeur
        'str'. On redirige sur __repr__"""

        return repr(self)

    def __len__(self):
        """Renvoie la taille du dictionnaire"""
        return len(self._cles)

    def __contains__(self, cle):
        """Renvoie True si la clé est dans la liste des clés, False sinon"""
        return cle in self._cles

    def __getitem__(self, cle):
        """Renvoie la valeur correspondant à la clé si elle existe, lève
        une exception KeyError sinon"""

        if cle not in self._cles:
            raise KeyError( \
                "La clé {0} ne se trouve pas dans le dictionnaire".format( \
                cle))
        else:
            indice = self._cles.index(cle)
            return self._valeurs[indice]

    def __setitem__(self, cle, valeur):
        """Méthode spéciale appelée quand on cherche à modifier une clé
        présente dans le dictionnaire. Si la clé n'est pas présente, on l'ajoute
        à la fin du dictionnaire"""

        if cle in self._cles:
            indice = self._cles.index(cle)
            self._valeurs[indice] = valeur
        else:
```

```
            self._cles.append(cle)
            self._valeurs.append(valeur)

    def __delitem__(self, cle):
        """Méthode appelée quand on souhaite supprimer une clé"""
        if cle not in self._cles:
            raise KeyError( \
                "La clé {0} ne se trouve pas dans le dictionnaire".format( \
                cle))
        else:
            indice = self._cles.index(cle)
            del self._cles[indice]
            del self._valeurs[indice]

    def __iter__(self):
        """Méthode de parcours de l'objet. On renvoie l'itérateur des clés"""
        return iter(self._cles)

    def __add__(self, autre_objet):
        """On renvoie un nouveau dictionnaire contenant les deux
        dictionnaires mis bout à bout (d'abord self puis autre_objet)"""

        if type(autre_objet) is not type(self):
            raise TypeError( \
                "Impossible de concaténer {0} et {1}".format( \
                type(self), type(autre_objet)))
        else:
            nouveau = DictionnaireOrdonne()

            # On commence par copier self dans le dictionnaire
            for cle, valeur in self.items():
                nouveau[cle] = valeur

            # On copie ensuite autre_objet
            for cle, valeur in autre_objet.items():
                nouveau[cle] = valeur
            return nouveau

    def items(self):
        """Renvoie un générateur contenant les couples (cle, valeur)"""
        for i, cle in enumerate(self._cles):
            valeur = self._valeurs[i]
            yield (cle, valeur)

```

```python
    def keys(self):
        """Cette méthode renvoie la liste des clés"""
        return list(self._cles)

    def values(self):
        """Cette méthode renvoie la liste des valeurs"""
        return list(self._valeurs)

    def reverse(self):
        """Inversion du dictionnaire"""
        # On crée deux listes vides qui contiendront le nouvel ordre des clés
        # et valeurs
        cles = []
        valeurs = []
        for cle, valeur in self.items():
            # On ajoute les clés et valeurs au début de la liste
            cles.insert(0, cle)
            valeurs.insert(0, valeur)
        # On met ensuite à jour nos listes
        self._cles = cles
        self._valeurs = valeurs

    def sort(self):
        """Méthode permettant de trier le dictionnaire en fonction de ses clés"""
        # On trie les clés
        cles_triees = sorted(self._cles)
        # On crée une liste de valeurs, encore vide
        valeurs = []
        # On parcourt ensuite la liste des clés triées
        for cle in cles_triees:
            valeur = self[cle]
            valeurs.append(valeur)
        # Enfin, on met à jour notre liste de clés et de valeurs
        self._cles = cles_triees
        self._valeurs = valeurs
```

Le mot de la fin

Le but de l'exercice était de présenter un énoncé simple et laissant de la place aux choix de programmation. Ce que je vous propose n'est pas l'unique façon de faire, ni la meilleure. L'exercice vous a surtout permis de travailler sur des notions concrètes que nous étudions depuis le début de cette partie et de construire un objet conteneur qui n'est pas dépourvu d'utilité.

N'hésitez pas à améliorer notre objet, il n'en sera que plus joli et utile avec des fonc-

tionnalités supplémentaires !

Ne vous alarmez pas si vous n'avez pas réussi à coder tous les aspects du dictionnaire. L'essentiel est d'avoir essayé et compris la correction.

Chapitre 23

Les décorateurs

Difficulté : 

Nous allons ici nous intéresser à un concept fascinant de Python, un concept de programmation assez avancé. Vous n'êtes pas obligés de lire ce chapitre pour la suite de ce livre, ni même connaître cette fonctionnalité pour coder en Python. Il s'agit d'un plus que j'ai voulu détailler mais qui n'est certainement pas indispensable.

Les décorateurs sont un moyen simple de modifier le comportement « par défaut » de fonctions. C'est un exemple assez flagrant de ce qu'on appelle la **métaprogrammation**, que je vais décrire assez brièvement comme l'écriture de programmes manipulant... d'autres programmes. Cela donne faim, non ?



Qu'est-ce que c'est ?

Les décorateurs sont des fonctions de Python dont le rôle est de modifier le comportement par défaut d'autres fonctions ou classes. Pour schématiser, une fonction modifiée par un décorateur ne s'exécutera pas elle-même mais appellera le décorateur. C'est au décorateur de décider s'il veut exécuter la fonction et dans quelles conditions.



Mais quel est l'intérêt ? Si on veut juste qu'une fonction fasse quelque chose de différent, il suffit de la modifier, non ? Pourquoi s'encombrer la tête avec une nouvelle fonctionnalité plus complexe ?

Il peut y avoir de nombreux cas dans lesquels les décorateurs sont un choix intéressant. Pour comprendre l'idée, je vais prendre un unique exemple.

On souhaite tester les performances de certaines de nos fonctions, en l'occurence, calculer combien de temps elles mettent pour s'exécuter.

Une possibilité, effectivement, consiste à modifier chacune des fonctions devant intégrer ce test. Mais ce n'est pas très élégant, ni très pratique, ni très sûr... bref ce n'est pas la meilleure solution.

Une autre possibilité consiste à utiliser un décorateur. Ce décorateur se chargera d'exécuter notre fonction en calculant le temps qu'elle met et pourra, par exemple, afficher une alerte si cette durée est trop élevée.

Pour indiquer qu'une fonction doit intégrer ce test, il suffira d'ajouter une simple ligne avant sa définition. C'est bien plus simple, clair et adapté à la situation.

Et ce n'est qu'un exemple d'application.

Les décorateurs sont des fonctions standard de Python mais leur construction est parfois complexe. Quand il s'agit de décorateurs prenant des arguments en paramètres ou devant tenir compte des paramètres de la fonction, le code est plus complexe, moins intuitif.

Je vais faire mon possible pour que vous compreniez bien le principe. N'hésitez pas à y revenir à tête reposée, une, deux, trois fois pour que cela soit bien clair.

En théorie

Une fois n'est pas coutume, je vais vous montrer les différentes constructions possibles en théorie avec quelques exemples, mais je vais aussi consacrer une section entière à des exemples d'utilisations pour expliciter cette partie théorique indispensable.

Format le plus simple

Comme je l'ai dit, les décorateurs sont des fonctions « classiques » de Python, dans leur définition. Ils ont une petite subtilité en ce qu'ils prennent en paramètre une fonction

et renvoient une fonction.

On déclare qu'une fonction doit être modifiée par un (ou plusieurs) décorateurs grâce à une (ou plusieurs) lignes au-dessus de la définition de fonction, comme ceci :

```
@nom_du_decorateur
def ma_fonction(...)
```

Le décorateur s'exécute au moment de la définition de fonction et non lors de l'appel. Ceci est important. Il prend en paramètre, comme je l'ai dit, une fonction (celle qu'il modifie) et renvoie une fonction (qui peut être la même).

Voyez plutôt :

```
>>> def mon_decorateur(fonction):
...     """Premier exemple de décorateur"""
...     print("Notre décorateur est appelé avec en paramètre la
    fonction {0}".format(fonction))
...     return fonction
...
>>> @mon_decorateur
... def salut():
...     """Fonction modifiée par notre décorateur"""
...     print("Salut !")
...
Notre décorateur est appelé avec en paramètre la fonction <
    function salut at 0x00BA5198>
>>>
```



Euh... qu'est-ce qu'on a fait là ?

- D'abord, on crée le décorateur. Il prend en paramètre, comme je vous l'ai dit, la fonction qu'il modifie. Dans notre exemple, il se contente d'afficher cette fonction puis de la renvoyer.
- On crée ensuite la fonction `salut`. Comme vous le voyez, on indique avant la définition la ligne `@mon_decorateur`, qui précise à Python que cette fonction doit être modifiée par notre décorateur. Notre fonction est très utile : elle affiche « Salut ! » et c'est tout.
- À la fin de la définition de notre fonction, on peut voir que le décorateur est appelé. Si vous regardez plus attentivement la ligne affichée, vous vous rendez compte qu'il est appelé avec, en paramètre, la fonction `salut` que nous venons de définir.

Intéressons-nous un peu plus à la structure de notre décorateur. Il prend en paramètre la fonction à modifier (celle que l'on définit sous la ligne du `@`), je pense que vous avez pu le constater. Mais il renvoie également cette fonction et cela, c'est un peu moins évident !

En fait, la fonction renvoyée remplace la fonction définie. Ici, on renvoie la fonction

définie, c'est donc la même. Mais on peut demander à Python d'exécuter une autre fonction à la place, pour modifier son comportement. Nous allons voir cela un peu plus loin.

Pour l'heure, souvenez-vous que les deux codes ci-dessous sont identiques :

```
# Exemple avec décorateur
@decorateur
def fonction(...):
    ...
```

```
# Exemple équivalent, sans décorateur
def fonction(...):
    ...

fonction = decorateur(fonction)
```

Relisez bien ces deux codes, ils font la même chose. Le second est là pour que vous compreniez ce que fait Python quand il manipule des fonctions modifiées par un (ou plusieurs) décorateur(s).

Quand vous exécutez `salut`, vous ne voyez aucun changement. Et c'est normal puisque nous renvoyons la même fonction. Le seul moment où notre décorateur est appelé, c'est lors de la définition de notre fonction. Notre fonction `salut` n'a pas été modifiée par notre décorateur, on s'est contenté de la renvoyer telle quelle.

Modifier le comportement de notre fonction

Vous l'aurez deviné, un décorateur comme nous l'avons créé plus haut n'est pas bien utile. Les décorateurs servent surtout à modifier le comportement d'une fonction. Je vous montre cependant pas à pas comment cela fonctionne, sinon vous risquez de vite vous perdre.



Comment faire pour modifier le comportement de notre fonction ?

En fait, vous avez un élément de réponse un peu plus haut. J'ai dit que notre décorateur prenait en paramètre la fonction définie et renvoyait une fonction (peut-être la même, peut-être une autre). C'est cette fonction renvoyée qui sera directement affectée à notre fonction définie. Si vous aviez renvoyé une autre fonction que `salut`, dans notre exemple ci-dessus, la fonction `salut` aurait redirigé vers cette fonction renvoyée.



Mais alors… il faut définir encore une fonction ?

Eh oui ! Je vous avais prévenus (et ce n'est que le début), notre construction se complexifie au fur et à mesure : on va devoir créer une nouvelle fonction qui sera chargée

de modifier le comportement de la fonction définie. Et, parce que notre décorateur sera le seul à utiliser cette fonction, on va la définir directement dans le corps de notre décorateur.



Je suis perdu. Comment cela marche-t-il, concrètement ?

Je vais vous mettre le code, cela vaudra mieux que des tonnes d'explications. Je le commente un peu plus bas, ne vous inquiétez pas :

```
def mon_decorateur(fonction):
    """Notre décorateur : il va afficher un message avant l'appel de la
    fonction définie"""

    def fonction_modifiee():
        """Fonction que l'on va renvoyer. Il s'agit en fait d'une version
        un peu modifiée de notre fonction originellement définie. On se
        contente d'afficher un avertissement avant d'exécuter notre fonction
        originellement définie"""

        print("Attention ! On appelle {0}".format(fonction))
        return fonction()
    return fonction_modifiee

@mon_decorateur
def salut():
    print("Salut !")
```

Voyons l'effet, avant les explications. Aucun message ne s'affiche en exécutant ce code. Par contre, si vous exécutez votre fonction `salut` :

```
>>> salut()
Attention ! On appelle <function salut at 0x00BA54F8>
Salut !
>>>
```

Et si vous affichez la fonction `salut` dans l'interpréteur, vous obtenez quelque chose de surprenant :

```
>>> salut
<function fonction_modifiee at 0x00BA54B0>
>>>
```

Pour comprendre, revenons sur le code de notre décorateur :

- Comme toujours, il prend en paramètre une fonction. Cette fonction, quand on place l'appel au décorateur au-dessus de `def salut`, c'est `salut` (la fonction définie à l'origine).
- Dans le corps même de notre décorateur, vous pouvez voir qu'on a défini une nouvelle fonction, `fonction_modifiee`. Elle ne prend aucun paramètre, elle n'en a pas besoin. Dans son corps, on affiche une ligne avertissant qu'on va exécuter la fonction `fonction` (là encore, il s'agit de `salut`). À la ligne suivante, on l'exécute effectivement et on renvoie le résultat de son exécution (dans le cas de `salut`, il n'y en a pas mais d'autres fonctions pourraient renvoyer des informations).
- De retour dans notre décorateur, on indique qu'il faut renvoyer `fonction_modifiee`.

Lors de la définition de notre fonction `salut`, on appelle notre décorateur. Python lui passe en paramètre la fonction `salut`. Cette fois, notre décorateur ne renvoie pas `salut` mais `fonction_modifiee`. Et notre fonction `salut`, que nous venons de définir, sera donc remplacée par notre fonction `fonction_modifiee`, définie dans notre décorateur.

Vous le voyez bien, d'ailleurs : quand on cherche à afficher `salut` dans l'interpréteur, on obtient `fonction_modifiee`.

Souvenez-vous bien que le code :

```
@mon_decorateur
def salut():
    ...
```

revient au même, pour Python, que le code :

```
def salut():
    ...

salut = mon_decorateur(salut)
```

Ce n'est peut-être pas plus clair. Prenez le temps de lire et de bien comprendre l'exemple. Ce n'est pas simple, la logique est bel et bien là mais il faut passer un certain temps à tester avant de bien intégrer cette notion.

Pour résumer, notre décorateur renvoie une fonction de substitution. Quand on appelle `salut`, on appelle en fait notre fonction modifiée qui appelle également `salut` après avoir affiché un petit message d'avertissement.

Autre exemple : un décorateur chargé tout simplement d'empêcher l'exécution de la fonction. Au lieu d'exécuter la fonction d'origine, on lève une exception pour avertir l'utilisateur qu'il utilise une fonctionnalité obsolète.

```
def obsolete(fonction_origine):
    """Décorateur levant une exception pour noter que la fonction_origine
    est obsolète"""

    def fonction_modifiee():
        raise RuntimeError("la fonction {0} est obsolète !".format(fonction_origine))
    return fonction_modifiee
```

Là encore, faites quelques essais : tout deviendra limpide après quelques manipulations.

Un décorateur avec des paramètres

Toujours plus dur ! On voudrait maintenant passer des paramètres à notre décorateur. Nous allons essayer de coder un décorateur chargé d'exécuter une fonction en contrôlant le temps qu'elle met à s'exécuter. Si elle met un temps supérieur à la durée passée en paramètre du décorateur, on affiche une alerte.

La ligne appelant notre décorateur, au-dessus de la définition de notre fonction, sera donc sous la forme :

```
@controler_temps(2.5) # 2,5 secondes maximum pour la fonction
    ci-dessous
```

Jusqu'ici, nos décorateurs ne comportaient aucune parenthèse après leur appel. Ces deux parenthèses sont très importantes : notre fonction de décorateur prendra en paramètres non pas une fonction, mais les paramètres du décorateur (ici, le temps maximum autorisé pour la fonction). Elle ne renverra pas une fonction de substitution, mais un décorateur.



Encore et toujours perdu. Pourquoi est-ce si compliqué de passer des paramètres à notre décorateur ?

En fait... ce n'est pas si compliqué que cela mais c'est dur à saisir au début. Pour mieux comprendre, essayez encore une fois de vous souvenir que ces deux codes reviennent au même :

```
@decorateur
def fonction(...):
    ...
```

```
def fonction(...):
    ...

fonction = decorateur(fonction)
```

C'est la dernière ligne du second exemple que vous devez retenir et essayer de comprendre : `fonction = decorateur(fonction)`.

On remplace la fonction que nous avons définie au-dessus par la fonction que renvoie notre décorateur.

C'est le mécanisme qui se cache derrière notre `@decorateur`.

Maintenant, si notre décorateur attend des paramètres, on se retrouve avec une ligne comme celle-ci :

```
@decorateur(parametre)
def fonction(...):
    ...
```

Et si vous avez compris l'exemple ci-dessus, ce code revient au même que :

```
def fonction(...):
    ...

fonction = decorateur(parametre)(fonction)
```

Je vous avais prévenus, ce n'est pas très intuitif ! Mais relisez bien ces exemples, le déclic devrait se faire tôt ou tard.

Comme vous le voyez, on doit définir comme décorateur une fonction qui prend en arguments les paramètres du décorateur (ici, le temps attendu) et qui renvoie un décorateur. Autrement dit, on se retrouve encore une fois avec un niveau supplémentaire dans notre fonction.

Je vous donne le code sans trop insister. Si vous arrivez à comprendre la logique qui se trouve derrière, c'est tant mieux, sinon n'hésitez pas à y revenir plus tard :

```
"""Pour gérer le temps, on importe le module time
On va utiliser surtout la fonction time() de ce module qui renvoie le nombre
de secondes écoulées depuis le premier janvier 1970 (habituellement).
On va s'en servir pour calculer le temps mis par notre fonction pour
s'exécuter"""

import time

def controler_temps(nb_secs):
    """Contrôle le temps mis par une fonction pour s'exécuter.
    Si le temps d'exécution est supérieur à nb_secs, on affiche une alerte"""

    def decorateur(fonction_a_executer):
        """Notre décorateur. C'est lui qui est appelé directement LORS
        DE LA DEFINITION de notre fonction (fonction_a_executer)"""

        def fonction_modifiee():
            """Fonction renvoyée par notre décorateur. Elle se charge
            de calculer le temps mis par la fonction à s'exécuter"""

            tps_avant = time.time() # Avant d'exécuter la fonction
            valeur_renvoyee = fonction_a_executer() # On exécute la fonction
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant
```

```
            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour s'exé
                    cuter".format( \
                        fonction_a_executer, tps_execution))
            return valeur_renvoyee
        return fonction_modifiee
    return decorateur
```

Ouf! Trois niveaux dans notre fonction! D'abord `controler_temps`, qui définit dans son corps notre décorateur `decorateur`, qui définit lui-même dans son corps notre fonction modifiée `fonction_modifiee`.

J'espère que vous n'êtes pas trop embrouillés. Je le répète, il s'agit d'une fonctionnalité très puissante mais qui n'est pas très intuitive quand on n'y est pas habitué. Jetez un coup d'œil du côté des exemples au-dessus si vous êtes un peu perdus.

Nous pouvons maintenant utiliser notre décorateur. J'ai fait une petite fonction pour tester qu'un message s'affiche bien si notre fonction met du temps à s'exécuter. Voyez plutôt :

```
>>> @controler_temps(4)
... def attendre():
...     input("Appuyez sur Entrée...")
...
>>> attendre() # Je vais appuyer sur Entrée presque tout de
    suite
Appuyez sur Entrée...
>>> attendre() # Cette fois, j'attends plus longtemps
Appuyez sur Entrée...
La fonction <function attendre at 0x00BA5810> a mis
    4.14100003242 pour s'exécuter
>>>
```

Ça marche! Et même si vous devez passer un peu de temps sur votre décorateur, vu ses différents niveaux, vous êtes obligés de reconnaître qu'il s'utilise assez simplement.

Il est quand même plus intuitif d'écrire :

```
@controler_temps(4)
def attendre(...)
    ...
```

que :

```
def attendre(...):
    ...

attendre = controler_temps(4)(attendre)
```

Tenir compte des paramètres de notre fonction

Jusqu'ici, nous n'avons travaillé qu'avec des fonctions ne prenant aucun paramètre. C'est pourquoi notre fonction `fonction_modifiee` n'en prenait pas non plus.

Oui mais… tenir compte des paramètres, cela peut être utile. Sans quoi on ne pourrait construire que des décorateurs s'appliquant à des fonctions sans paramètre.

Il faut, pour tenir compte des paramètres de la fonction, modifier ceux de notre fonction `fonction_modifiee`. Là encore, je vous invite à regarder les exemples ci-dessus, explicitant ce que Python fait réellement lorsqu'on définit un décorateur avant une fonction. Vous pourrez vous rendre compte que `fonction_modifiee` remplace notre fonction et que, par conséquent, elle doit prendre des paramètres si notre fonction définie prend également des paramètres.

C'est dans ce cas en particulier que nous allons pouvoir réutiliser la notation spéciale pour nos fonctions attendant un nombre variable d'arguments. En effet, le décorateur que nous avons créé un peu plus haut devrait pouvoir s'appliquer à des fonctions ne prenant aucun paramètre, ou en prenant un, ou plusieurs… au fond, notre décorateur ne doit ni savoir combien de paramètres sont fournis à notre fonction, ni même s'en soucier.

Là encore, je vous donne le code adapté de notre fonction modifiée. Souvenez-vous qu'elle est définie dans notre `decorateur`, lui-même défini dans `controler_temps` (je ne vous remets que le code de `fonction_modifiee`).

```
...
        def fonction_modifiee(*parametres_non_nommes, **parametres_nommes):
            """Fonction renvoyée par notre décorateur. Elle se charge
            de calculer le temps mis par la fonction à s'exécuter"""

            tps_avant = time.time() # avant d'exécuter la fonction
            ret = fonction_a_executer(*parametres_non_nommes, **parametres_nommes)
            tps_apres = time.time()
            tps_execution = tps_apres - tps_avant
            if tps_execution >= nb_secs:
                print("La fonction {0} a mis {1} pour s'exécuter".format( \
                        fonction_a_executer, tps_execution))
            return ret
```

À présent, vous pouvez appliquer ce décorateur à des fonctions ne prenant aucun paramètre, ou en prenant un certain nombre, nommés ou non. Pratique, non ?

Des décorateurs s'appliquant aux définitions de classes

Vous pouvez également appliquer des décorateurs aux définitions de classes. Nous verrons un exemple d'application dans la section suivante. Au lieu de recevoir en paramètre la fonction, vous allez recevoir la classe.

```
>>> def decorateur(classe):
...     print("Définition de la classe {0}".format(classe))
...     return classe
...
>>> @decorateur
... class Test:
...     pass
...
Définition de la classe <class '__main__.Test'>
>>>
```

Voilà. Vous verrez dans la section suivante quel peut être l'intérêt de manipuler nos définitions de classes à travers des décorateurs. Il existe d'autres exemples que celui que je vais vous montrer, bien entendu.

Chaîner nos décorateurs

Vous pouvez modifier une fonction ou une définition de classe par le biais de plusieurs décorateurs, sous la forme :

```
@decorateur1
@decorateur2
def fonction():
```

Ce n'est pas plus compliqué que ce que vous venez de faire. Je vous le montre pour qu'il ne subsiste aucun doute dans votre esprit, vous pouvez tester à loisir cette possibilité, par vous-mêmes.

Je vais à présent vous présenter quelques applications possibles des décorateurs, inspirées en grande partie de la PEP 318, accessible avec le code web suivant :

▷ PEP 318
Code web : 647141

Exemples d'applications

Nous allons voir deux exemples d'applications des décorateurs dans cette section. Vous en avez également vu quelques-uns dans la section précédente mais, maintenant que vous maîtrisez la syntaxe, nous allons nous pencher sur des exemples plus parlants !

Les classes singleton

Certains reconnaîtront sûrement cette appellation. Pour les autres, sachez qu'une classe dite `singleton` est une classe qui ne peut être instanciée qu'une fois.

Autrement dit, on ne peut créer qu'un seul objet de cette classe.

Cela peut-être utile quand vous voulez être absolument certains qu'une classe ne produira qu'un seul objet, qu'il est inutile (voire dangereux) d'avoir plusieurs objets de cette classe. La première fois que vous appelez le constructeur de ce type de classe, vous obtenez le premier et l'unique objet nouvellement instancié. Tout appel ultérieur à ce constructeur renvoie le même objet (le premier créé).

Ceci est très facile à modéliser grâce à des décorateurs.

Code de l'exemple

```
def singleton(classe_definie):
    instances = {} # Dictionnaire de nos instances singletons
    def get_instance():
        if classe_definie not in instances:
            # On crée notre premier objet de classe_definie
            instances[classe_definie] = classe_definie()
        return instances[classe_definie]
    return get_instance
```

Explications

D'abord, pour utiliser notre décorateur, c'est très simple : il suffit de mettre l'appel à notre décorateur avant la définition des classes que nous souhaitons utiliser en tant que `singleton` :

```
>>> @singleton
... class Test:
...     pass
...
>>> a = Test()
>>> b = Test()
>>> a is b
True
>>>
```

Quand on crée notre premier objet (celui se trouvant dans `a`), notre constructeur est bien appelé. Quand on souhaite créer un second objet, c'est celui contenu dans `a` qui est renvoyé. Ainsi, `a` et `b` pointent vers le même objet.

Intéressons-nous maintenant à notre décorateur. Il définit dans son corps un dictionnaire. Ce dictionnaire contient en guise de clé la classe `singleton` et en tant que valeur l'objet créé correspondant. Il renvoie notre fonction interne `get_instance` qui va remplacer notre classe. Ainsi, quand on voudra créer un nouvel objet, ce sera `get_instance`

qui sera appelée. Cette fonction vérifie si notre classe se trouve dans le dictionnaire. Si ce n'est pas le cas, on crée notre premier objet correspondant et on l'insère dans le dictionnaire. Dans tous les cas, on renvoie l'objet correspondant dans le dictionnaire (soit il vient d'être créé, soit c'est notre objet créé au premier appel du constructeur).

Grâce à ce système, on peut avoir plusieurs classes déclarées comme des `singleton` et on est sûr que, pour chacune de ces classes, *un seul* objet sera créé.

Contrôler les types passés à notre fonction

Vous l'avez déjà observé dans Python : aucun contrôle n'est fait sur le type des données passées en paramètres de nos fonctions. Certaines, comme `print`, acceptent n'importe quel type. D'autres lèvent des exceptions quand un paramètre d'un type incorrect leur est fourni.

Il pourrait être utile de coder un décorateur qui vérifie les types passés en paramètres à notre fonction et qui lève une exception si les types attendus ne correspondent pas à ceux reçus lors de l'appel à la fonction.

Voici notre définition de fonction, pour vous donner une idée :

```
@controler_types(int, int)
def intervalle(base_inf, base_sup):
```

Notre décorateur `controler_types` doit s'assurer qu'à chaque fois qu'on appelle la fonction `intervalle`, ce sont des entiers qui sont passés en paramètres en tant que `base_inf` et `base_sup`.

Ce décorateur est plus complexe, bien que j'aie simplifié au maximum l'exemple de la PEP 318.

Encore une fois, s'il est un peu long à écrire, il est d'une simplicité enfantine à utiliser.

Code de l'exemple

```
def controler_types(*a_args, **a_kwargs):
    """On attend en paramètres du décorateur les types souhaités. On accepte
    une liste de paramètres indéterminés, étant donné que notre fonction
    définie pourra être appelée avec un nombre variable de paramètres et que
    chacun doit être contrôlé"""

    def decorateur(fonction_a_executer):
        """Notre décorateur. Il doit renvoyer fonction_modifiee"""
        def fonction_modifiee(*args, **kwargs):
            """Notre fonction modifiée. Elle se charge de contrôler
```

```
            les types qu'on lui passe en paramètres"""

            # La liste des paramètres attendus (a_args) doit être de même
            # Longueur que celle reçue (args)
            if len(a_args) != len(args):
                raise TypeError("le nombre d'arguments attendu n'est pas égal " \
                        "au nombre reçu")
            # On parcourt la liste des arguments reçus et non nommés
            for i, arg in enumerate(args):
                if a_args[i] is not type(args[i]):
                    raise TypeError("l'argument {0} n'est pas du type " \
                            "{1}".format(i, a_args[i]))

            # On parcourt à présent la liste des paramètres reçus et nommés
            for cle in kwargs:
                if cle not in a_kwargs:
                    raise TypeError("l'argument {0} n'a aucun type " \
                            "précisé".format(repr(cle)))
                if a_kwargs[cle] is not type(kwargs[cle]):
                    raise TypeError("l'argument {0} n'est pas de type" \
                            "{1}".format(repr(cle), a_kwargs[cle]))
            return fonction_a_executer(*args, **kwargs)
        return fonction_modifiee
    return decorateur
```

Explications

C'est un décorateur assez complexe (et pourtant, croyez-moi, je l'ai simplifié autant que possible). Nous allons d'abord voir comment l'utiliser :

```
>>> @controler_types(int, int)
... def intervalle(base_inf, base_sup):
...     print("Intervalle de {0} à {1}".format(base_inf, base_sup))
...
>>> intervalle(1, 8)
Intervalle de 1 à 8
>>> intervalle(5, "oups!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 24, in fonction_modifiee
TypeError: l'argument 1 n'est pas du type <class 'int'>
```

```
>>>
```

Là encore, l'utilisation est des plus simples. Intéressons-nous au décorateur proprement dit, c'est déjà un peu plus complexe.

Notre décorateur doit prendre des paramètres (une liste de paramètres indéterminés d'ailleurs, car notre fonction doit elle aussi prendre une liste de paramètres indéterminés et l'on doit contrôler chacun d'eux). On définit donc un paramètre `a_args` qui contient la liste des types des paramètres non nommés attendus, et un second paramètre `a_kwargs` qui contient le dictionnaire des types des paramètres nommés attendus.

Vous suivez toujours ?

Vous devriez comprendre la construction d'ensemble, nous l'avons vue un peu plus haut. Elle comprend trois niveaux, puisque nous devons influer sur le comportement de la fonction et que notre décorateur prend des paramètres. Notre code de contrôle se trouve, comme il se doit, dans notre fonction `fonction_modifiee` (qui va prendre la place de notre `fonction_a_executer`).

On commence par vérifier que la liste des paramètres non nommés attendus est bien égale en taille à la liste des paramètres non nommés reçus. On vérifie ensuite individuellement chaque paramètre reçu, en contrôlant son type. Si le type reçu est égal au type attendu, tout va bien. Sinon, on lève une exception. On répète l'opération sur les paramètres nommés (avec une petite différence, puisqu'il s'agit de paramètres nommés : ils sont contenus dans un dictionnaire, pas une liste).

Si tout va bien (aucune exception n'a été levée), on exécute notre fonction en renvoyant son résultat.

Voilà nos exemples d'applications. Il y en a bien d'autres, vous pouvez en retrouver plusieurs sur la PEP 318 consacrée aux décorateurs, ainsi que des informations supplémentaires : n'hésitez pas à y faire un petit tour.

En résumé

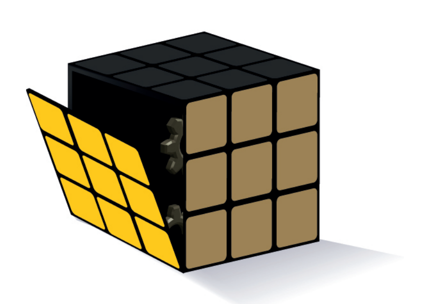
- Les décorateurs permettent de modifier le comportement d'une fonction.
- Ce sont eux-mêmes des fonctions, prenant en paramètre une fonction et renvoyant une fonction (qui peut être la même).
- On peut déclarer une fonction comme décorée en plaçant, au-dessus de la ligne de sa définition, la ligne `@nom_decorateur`.
- Au moment de la définition de la fonction, le décorateur est appelé et la fonction qu'il renvoie sera celle utilisée.
- Les décorateurs peuvent également prendre des paramètres pour influer sur le comportement de la fonction décorée.

Chapitre 24

Les métaclasses

Difficulté : 

Toujours plus loin vers la métaprogrammation ! Nous allons ici nous intéresser au concept des métaclasses, ou comment générer des classes à partir. . . d'autres classes ! Je ne vous cache pas qu'il s'agit d'un concept assez avancé de la programmation Python, prenez donc le temps nécessaire pour comprendre ce nouveau concept.



Retour sur le processus d'instanciation

Depuis la troisième partie de ce cours, nous avons créé bon nombre d'objets. Nous avons découvert au début de cette partie le **constructeur**, cette méthode appelée quand on souhaite créer un objet.

Je vous ai dit alors que les choses étaient un peu plus complexes que ce qu'il semblait. Nous allons maintenant voir en quoi !

Admettons que vous ayez défini une classe :

```
class Personne:

    """Classe définissant une personne.

    Elle possède comme attributs :
    nom -- le nom de la personne
    prenom -- son prénom
    age -- son âge
    lieu_residence -- son lieu de résidence

    Le nom et le prénom doivent être passés au constructeur."""

    def __init__(self, nom, prenom):
        """Constructeur de notre personne."""
        self.nom = nom
        self.prenom = prenom
        self.age = 23
        self.lieu_residence = "Lyon"
```

Cette syntaxe n'a rien de nouveau pour nous.

Maintenant, que se passe-t-il quand on souhaite créer une personne ? Facile, on rédige le code suivant :

```
personne = Personne("Doe", "John")
```

Lorsque l'on exécute cela, Python appelle notre constructeur `__init__` en lui transmettant les arguments fournis à la construction de l'objet. Il y a cependant une étape intermédiaire.

Si vous examinez la définition de notre constructeur :

```
def __init__(self, nom, prenom):
```

Vous ne remarquez rien d'étrange ? Peut-être pas, car vous avez été habitués à cette syntaxe depuis le début de cette partie : la méthode prend en premier paramètre `self`.

Or, `self`, vous vous en souvenez, c'est l'objet que nous manipulons. Sauf que, quand on crée un objet... on souhaite récupérer un nouvel objet mais on n'en passe aucun à la classe.

D'une façon ou d'une autre, notre classe crée un nouvel objet et le passe à notre constructeur. La méthode `__init__` se charge d'écrire dans notre objet ses attributs,

mais elle n'est pas responsable de la création de notre objet. Nous allons à présent voir qui s'en charge.

La méthode `__new__`

La méthode `__init__`, comme nous l'avons vu, est là pour *initialiser* notre objet (en écrivant des attributs dedans, par exemple) mais elle n'est pas là pour le *créer*. La méthode qui s'en charge, c'est `__new__`.

C'est aussi une méthode spéciale, vous en reconnaissez la particularité. C'est également une méthode définie par `object`, que l'on peut redéfinir en cas de besoin.

Avant de voir ce qu'elle prend en paramètres, voyons plus précisément ce qui se passe quand on tente de construire un objet :

- On demande à créer un objet, en écrivant par exemple `Personne("Doe", "John")`.
- La méthode `__new__` de notre classe (ici `Personne`) est appelée et se charge de construire un nouvel objet.
- Si `__new__` renvoie une instance de la classe, on appelle le constructeur `__init__` en lui passant en paramètres cette nouvelle instance ainsi que les arguments passés lors de la création de l'objet.

Maintenant, intéressons-nous à la structure de notre méthode `__new__`.

C'est une méthode statique, ce qui signifie qu'elle ne prend pas `self` en paramètre. C'est logique, d'ailleurs : son but est de créer une nouvelle instance de classe, l'instance n'existe pas encore.

Elle ne prend donc pas `self` en premier paramètre (l'instance d'objet). Cependant, elle prend la classe manipulée `cls`.

Autrement dit, quand on souhaite créer un objet de la classe `Personne`, la méthode `__new__` de la classe `Personne` est appelée et prend comme premier paramètre la classe `Personne` elle-même.

Les autres paramètres passés à la méthode `__new__` seront transmis au constructeur.

Voyons un peu cela, exprimé sous forme de code :

```
class Personne:

    """Classe définissant une personne.

    Elle possède comme attributs :
    nom -- le nom de la personne
    prenom -- son prénom
    age -- son âge
    lieu_residence -- son lieu de résidence

    Le nom et le prénom doivent être passés au constructeur."""

    def __new__(cls, nom, prenom):
        print("Appel de la méthode __new__ de la classe {}".
```

```
                format(cls))
        # On laisse le travail à object
        return object.__new__(cls, nom, prenom)

    def __init__(self, nom, prenom):
        """Constructeur de notre personne."""
        print("Appel de la méthode __init__")
        self.nom = nom
        self.prenom = prenom
        self.age = 23
        self.lieu_residence = "Lyon"
```

Essayons de créer une personne :

```
>>> personne = Personne("Doe", "John")
Appel de la méthode __new__ de la classe <class '__main__.
    Personne'>
Appel de la méthode __init__
>>>
```

Redéfinir `__new__` peut permettre, par exemple, de créer une instance d'une autre classe. Elle est principalement utilisée par Python pour produire des types **immuables** (en anglais, *immutable*), que l'on ne peut modifier, comme le sont les chaînes de caractères, les tuples, les entiers, les flottants...

La méthode `__new__` est parfois redéfinie dans le corps d'une métaclasse. Nous allons à présent voir de ce dont il s'agit.

Créer une classe dynamiquement

Je le répète une nouvelle fois, *en Python, tout est objet.* Cela veut dire que les entiers, les flottants, les listes sont des objets, que les modules sont des objets, que les packages sont des objets... mais cela veut aussi dire que les classes sont des objets !

La méthode que nous connaissons

Pour créer une classe, nous n'avons vu qu'une méthode, la plus utilisée, faisant appel au mot-clé `class`.

```
class MaClasse:
```

Vous pouvez ensuite créer des instances sur le modèle de cette classe, je ne vous apprends rien.

Mais là où cela se complique, c'est que les classes sont également des objets.



Si les classes sont des objets... cela veut dire que les classes sont elles-mêmes modelées sur des classes ?

Eh oui. Les classes, comme tout objet, sont modelées sur une classe. Cela paraît assez difficile à comprendre au début. Peut-être cet extrait de code vous aidera-t-il à comprendre l'idée.

```
>>> type(5)
<class 'int'>
>>> type("une chaîne")
<class 'str'>
>>> type([1, 2, 3])
<class 'list'>
>>> type(int)
<class 'type'>
>>> type(str)
<class 'type'>
>>> type(list)
<class 'type'>
>>>
```

On demande le type d'un entier et Python nous répond `class int`. Sans surprise. Mais si on lui demande la classe de `int`, Python nous répond `class type`.

En fait, par défaut, toutes nos classes sont modelées sur la classe `type`. Cela signifie que :

1. quand on crée une nouvelle classe (`class Personne:` par exemple), Python appelle la méthode `__new__` de la classe `type` ;
2. une fois la classe créée, on appelle le constructeur `__init__` de la classe `type`.

Cela semble sans doute encore obscur. Ne désespérez pas, vous comprendrez peut-être un peu mieux ce dont je parle en lisant la suite. Sinon, n'hésitez pas à relire ce passage et à faire des tests par vous-mêmes.

Créer une classe dynamiquement

Résumons :

- nous savons que les objets sont modelés sur des classes ;
- nous savons que nos classes, étant elles-mêmes des objets, sont modelées sur une classe ;
- la classe sur laquelle toutes les autres sont modelées par défaut s'appelle `type`.

Je vous propose d'essayer de créer une classe dynamiquement, sans passer par le mot-clé `class` mais par la classe `type` directement.

La classe `type` prend trois arguments pour se construire :

- le nom de la classe à créer ;
- un **tuple** contenant les classes dont notre nouvelle classe va hériter ;
- un dictionnaire contenant les attributs et méthodes de notre classe.

```
>>> Personne = type("Personne", (), {})
>>> Personne
<class '__main__.Personne'>
>>> john = Personne()
>>> dir(john)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__g
e__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '_
_setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>>
```

J'ai simplifié le code au maximum. Nous créons bel et bien une nouvelle classe que nous stockons dans notre variable `Personne`, mais elle est vide. Elle n'hérite d'aucune classe et elle ne définit aucun attribut ni méthode de classe.

Nous allons essayer de créer deux méthodes pour notre classe :

- un constructeur `__init__` ;
- une méthode `presenter` affichant le prénom et le nom de la personne.

Je vous donne ici le code auquel on peut arriver :

```
def creer_personne(personne, nom, prenom):
    """La fonction qui jouera le rôle de constructeur pour notre classe Personne.

    Elle prend en paramètre, outre la personne :
    nom -- son nom
    prenom -- son prenom"""

    personne.nom = nom
    personne.prenom = prenom
    personne.age = 21
    personne.lieu_residence = "Lyon"

def presenter_personne(personne):
    """Fonction présentant la personne.

    Elle affiche son prénom et son nom"""

    print("{} {}".format(personne.prenom, personne.nom))

# Dictionnaire des méthodes
methodes = {
```

```
    "__init__": creer_personne,
    "presenter": presenter_personne,
}

# Création dynamique de la classe
Personne = type("Personne", (), methodes)
```

Avant de voir les explications, voyons les effets :

```
>>> john = Personne("Doe", "John")
>>> john.nom
'Doe'
>>> john.prenom
'John'
>>> john.age
21
>>> john.presenter()
John Doe
>>>
```

Je ne vous le cache pas, c'est une fonctionnalité que vous utiliserez sans doute assez rarement. Mais cette explication était à propos quand on s'intéresse aux métaclasses.

Pour l'heure, décomposons notre code :

1. On commence par créer deux fonctions, `creer_personne` et `presenter_personne`. Elles sont amenées à devenir les méthodes `__init__` et `presenter` de notre future classe. Étant de futures méthodes d'instance, elles doivent prendre en premier paramètre l'objet manipulé.
2. On place ces deux fonctions dans un dictionnaire. En clé se trouve le nom de la future méthode et en valeur, la fonction correspondante.
3. Enfin, on fait appel à `type` en lui passant, en troisième paramètre, le dictionnaire que l'on vient de constituer.

Si vous essayez de mettre des attributs dans ce dictionnaire passé à `type`, vous devez être conscients du fait qu'il s'agira d'attributs de classe, pas d'attributs d'instance.

Définition d'une métaclasse

Nous avons vu que `type` est la métaclasse de toutes les classes par défaut. Cependant, une classe peut posséder une autre métaclasse que `type`.

Construire une métaclasse se fait de la même façon que construire une classe. Les métaclasses héritent de `type`. Nous allons retrouver la structure de base des classes que nous avons vues auparavant.

Nous allons notamment nous intéresser à deux méthodes que nous avons utilisées dans nos définitions de classes :

– la méthode `__new__`, appelée pour créer une classe ;
– la méthode `__init__`, appelée pour construire la classe.

La méthode `__new__`

Elle prend quatre paramètres :

– la métaclasse servant de base à la création de notre nouvelle classe ;
– le nom de notre nouvelle classe ;
– un **tuple** contenant les classes dont héritent notre classe à créer ;
– le dictionnaire des attributs et méthodes de la classe à créer.

Les trois derniers paramètres, vous devriez les reconnaître : ce sont les mêmes que ceux passés à `type`.

Voici une méthode `__new__` minimaliste.

```
class MaMetaClasse(type):

    """Exemple d'une métaclasse."""

    def __new__(metacls, nom, bases, dict):
        """Création de notre classe."""
        print("On crée la classe {}".format(nom))
        return type.__new__(metacls, nom, bases, dict)
```

Pour dire qu'une classe prend comme métaclasse autre chose que `type`, c'est dans la ligne de la définition de la classe que cela se passe :

```
class MaClasse(metaclass=MaMetaClasse):
    pass
```

En exécutant ce code, vous pouvez voir :

```
On crée la classe MaClasse
```

La méthode `__init__`

Le constructeur d'une métaclasse prend les mêmes paramètres que `__new__`, sauf le premier, qui n'est plus la métaclasse servant de modèle mais la classe que l'on vient de créer.

Les trois paramètres suivants restent les mêmes : le nom, le **tuple** des classes-mères et le dictionnaire des attributs et méthodes de classe.

Il n'y a rien de très compliqué dans le procédé, l'exemple ci-dessus peut être repris en le modifiant quelque peu pour qu'il s'adapte à la méthode `__init__`.

Maintenant, voyons concrètement à quoi cela peut servir.

Les métaclasses en action

Comme vous pouvez vous en douter, les métaclasses sont généralement utilisées pour des besoins assez complexes. L'exemple le plus répandu est une métaclasse chargée de tracer l'appel de ses méthodes. Autrement dit, dès qu'on appelle une méthode d'un objet, une ligne s'affiche pour le signaler. Mais cet exemple est assez difficile à comprendre car il fait appel à la fois au concept des métaclasses et à celui des décorateurs, pour décorer les méthodes tracées.

Je vous propose quelque chose de plus simple. Il va de soi qu'il existe bien d'autres usages, dont certains complexes, des métaclasses.

Nous allons essayer de garder nos classes créées dans un dictionnaire prenant comme clé le nom de la classe et comme valeur la classe elle-même.

Par exemple, dans une bibliothèque destinée à construire des interfaces graphiques, on trouve plusieurs *widgets*[1] comme des boutons, des cases à cocher, des menus, des cadres... Généralement, ces objets sont des classes héritant d'une classe mère commune. En outre, l'utilisateur peut, en cas de besoin, créer ses propres classes héritant des classes de la bibliothèque.

Par exemple, la classe mère de tous nos widgets s'appellera `Widget`. De cette classe hériteront les classes `Bouton`, `CaseACocher`, `Menu`, `Cadre`, etc. L'utilisateur de la bibliothèque pourra par ailleurs en dériver ses propres classes.

Le dictionnaire que l'on aimerait créer se présente comme suit :

```
{
    "Widget": Widget,
    "Bouton": Bouton,
    "CaseACocher": CaseACocher,
    "Menu": Menu,
    "Cadre": Cadre,
    ...
}
```

Ce dictionnaire pourrait être rempli manuellement à chaque fois qu'on crée une classe héritant de `Widget` mais avouez que ce ne serait pas très pratique.

Dans ce contexte, les métaclasses peuvent nous faciliter la vie. Vous pouvez essayer de faire l'exercice, le code n'est pas trop complexe. Cela dit, étant donné qu'on a vu beaucoup de choses dans ce chapitre et que les métaclasses sont un concept plutôt avancé, je vous donne directement le code qui vous aidera peut-être à comprendre le mécanisme :

```
trace_classes = {} # Notre dictionnaire vide

class MetaWidget(type):

    """Notre métaclasse pour nos Widgets.

```

1. Ce sont des objets graphiques.

```python
    Elle hérite de type, puisque c'est une métaclasse.
    Elle va écrire dans le dictionnaire trace_classes à chaque fois
    qu'une classe sera créée, utilisant cette métaclasse naturellement."""

    def __init__(cls, nom, bases, dict):
        """Constructeur de notre métaclasse, appelé quand on crée une classe."""
        type.__init__(cls, nom, bases, dict)
        trace_classes[nom] = cls
```

Pas trop compliqué pour l'heure. Créons notre classe `Widget` :

```python
class Widget(metaclass=MetaWidget):

    """Classe mère de tous nos widgets."""

    pass
```

Après avoir exécuté ce code, vous pouvez voir que notre classe `Widget` a bien été ajoutée dans notre dictionnaire :

```python
>>> trace_classes
{'Widget': <class '__main__.Widget'>}
>>>
```

Maintenant, construisons une nouvelle classe héritant de `Widget`.

```python
class bouton(Widget):

    """Une classe définissant le widget bouton."""

    pass
```

Si vous affichez de nouveau le contenu du dictionnaire, vous vous rendrez compte que la classe `Bouton` a bien été ajoutée. Héritant de `Widget`, elle reprend la même métaclasse (sauf mention contraire explicite) et elle est donc ajoutée au dictionnaire.

Vous pouvez étoffer cet exemple, faire en sorte que l'aide de la classe soit également conservée, ou qu'une exception soit levée si une classe du même nom existe déjà dans le dictionnaire.

Pour conclure

Les métaclasses sont un concept de programmation assez avancé, puissant mais délicat à comprendre de prime abord. Je vous invite, en cas de doute, à tester par vous-mêmes ou à rechercher d'autres exemples, ils sont nombreux.

En résumé

- Le processus d'instanciation d'un objet est assuré par deux méthodes, `__new__` et `__init__`.
- `__new__` est chargée de la création de l'objet et prend en premier paramètre sa classe.
- `__init__` est chargée de l'initialisation des attributs de l'objet et prend en premier paramètre l'objet précédemment créé par `__new__`.
- Les classes étant des objets, elles sont toutes modelées sur une classe appelée **métaclasse**.
- À moins d'être explicitement modifiée, la métaclasse de toutes les classes est `type`.
- On peut utiliser `type` pour créer des classes dynamiquement.
- On peut faire hériter une classe de `type` pour créer une nouvelle métaclasse.
- Dans le corps d'une classe, pour spécifier sa métaclasse, on exploite la syntaxe suivante : `class MaClasse(metaclass=NomDeLaMetaClasse):`.

Quatrième partie

Les merveilles de la bibliothèque standard

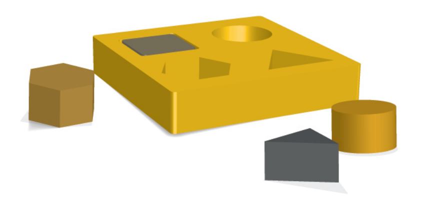
Chapitre 25

Les expressions régulières

Difficulté : 

Dans ce chapitre, je vais m'attarder sur les **expressions régulières** et sur le module `re` qui permet de les manipuler. En quelques mots, sachez que les expressions régulières permettent de réaliser très rapidement et facilement des recherches sur des chaînes de caractères.

Il existe, naturellement, bien d'autres modules permettant de manipuler du texte. C'est toutefois sur celui-ci que je vais m'attarder aujourd'hui, tout en vous donnant les moyens d'aller plus loin si vous le désirez.



Que sont les expressions régulières ?

Les **expressions régulières** sont un puissant moyen de rechercher et d'isoler des expressions d'une chaîne de caractères.

Pour simplifier, imaginez que vous faites un programme qui demande un certain nombre d'informations à l'utilisateur afin de les stocker dans un fichier. Lui demander son nom, son prénom et quelques autres informations, ce n'est pas bien difficile : on va utiliser la fonction `input` et récupérer le résultat. Jusqu'ici, rien de nouveau.

Mais si on demande à l'utilisateur de fournir un numéro de téléphone ? Qu'est-ce qui l'empêche de taper n'importe quoi ? Si on lui demande de fournir une adresse e-mail et qu'il tape quelque chose d'invalide, par exemple « je_te_donnerai_pas_mon_email », que va-t-il se passer si l'on souhaite envoyer automatiquement un email à cette personne ?

Si ce cas n'est pas géré, vous risquez d'avoir un problème. Les expressions régulières sont un moyen de rechercher, d'isoler ou de remplacer des expressions dans une chaîne. Ici, elles nous permettraient de vérifier que le numéro de téléphone saisi compte bien dix chiffres, qu'il commence par un 0 et qu'il compte éventuellement des séparateurs tous les deux chiffres. Si ce n'est pas le cas, on demande à l'utilisateur de le saisir à nouveau.

Quelques éléments de syntaxe pour les expressions régulières

Si vous connaissez déjà les expressions régulières et leur syntaxe, vous pouvez passer directement à la section consacrée au module `re`. Sinon, sachez que je ne pourrai vous présenter que brièvement les expressions régulières. C'est un sujet très vaste, qui mérite un livre à lui tout seul. Ne paniquez pas, toutefois, je vais vous donner quelques exemples concrets et vous pourrez toujours trouvez des explications plus approfondies de par le Web.

Concrètement, comment cela se présente-t-il ?

Le module `re`, que nous allons découvrir un peu plus loin, nous permet de faire des recherches très précises dans des chaînes de caractères et de remplacer des éléments de nos chaînes, le tout en fonction de critères particuliers. Ces critères, ce sont nos **expressions régulières**. Pour nous, elles se présentent sous la forme de chaînes de caractères. Les expressions régulières deviennent assez rapidement difficiles à lire mais ne vous en faites pas : nous allons y aller petit à petit.

Des caractères ordinaires

Quand on forme une expression régulière, on peut utiliser des caractères spéciaux et d'autres qui ne le sont pas. Par exemple, si nous recherchons le mot `chat` dans notre

chaîne, nous pouvons écrire comme expression régulière la chaîne « `chat` ». Jusque là, rien de très compliqué.

Mais vous vous doutez bien que les expressions régulières ne se limitent pas à ce type de recherche extrêmement simple, sans quoi les méthodes `find` et `replace` de la classe `str` auraient suffit.

Rechercher au début ou à la fin de la chaîne

Vous pouvez rechercher au début de la chaîne en plaçant en tête de votre regex[1] le signe d'accent circonflexe `^`. Si, par exemple, vous voulez rechercher la syllabe `cha` en début de votre chaîne, vous écrirez donc l'expression `^cha`. Cette expression sera trouvée dans la chaîne `'chaton'` mais pas dans la chaîne `'achat'`.

Pour matérialiser la fin de la chaîne, vous utiliserez le signe `$`. Ainsi, l'expression `q$` sera trouvée uniquement si votre chaîne se termine par la lettre q minuscule.

Contrôler le nombre d'occurences

Les caractères spéciaux que nous allons découvrir permettent de contrôler le nombre de fois où notre expression apparaît dans notre chaîne.

Regardez l'exemple ci-dessous :

```
chat*
```

Nous avons rajouté un astérisque (`*`) après le caractère `t` de `chat`. Cela signifie que notre lettre `t` pourra se retrouver 0, 1, 2, ... fois dans notre chaîne. Autrement dit, notre expression `chat*` sera trouvée dans les chaînes suivantes : `'chat'`, `'chaton'`, `'chateau'`, `'herbe à chat'`, `'chapeau'`, `'chatterton'`, `'chattttttttt'`...

Regardez un à un les exemples ci-dessus pour vérifier que vous les comprenez bien. On trouvera dans chacune de ces chaînes l'expression régulière `chat*`. Traduite en français, cette expression signifie : « on recherche une lettre `c` suivie d'une lettre `h` suivie d'une lettre `a` suivie, éventuellement, d'une lettre `t` qu'on peut trouver zéro, une ou plusieurs fois ». Peu importe que ces lettres soient trouvées au début, à la fin ou au milieu de la chaîne.

Un autre exemple ? Considérez l'expression régulière ci-dessous et essayez de la comprendre :

```
bat*e
```

Cette expression est trouvée dans les chaînes suivantes : `'bateau'`, `'batteur'` et `'joan baez'`.

Dans nos exemples, le signe `*` n'agit que sur la lettre qui le précède directement, pas sur les autres lettres qui figurent avant ou après.

1. Abréviation de *Regular Expression*.

Il existe d'autres signes permettant de contrôler le nombre d'occurences d'une lettre. Je vous ai fait un petit récapitulatif dans le tableau suivant, en prenant des exemples d'expressions avec les lettres `a`, `b` et `c` :

Signe	**Explication**	**Expression**	**Chaînes contenant l'expression**
*	0, 1 ou plus	abc*	'ab', 'abc', 'abcc', 'abcccccc'
+	1 ou plus	abc+	'abc', 'abcc', 'abccc'
?	0 ou 1	abc?	'ab', 'abc'

Vous pouvez également contrôler précisément le nombre d'occurences grâce aux accolades :

- `E{4}` : signifie 4 fois la lettre E majuscule ;
- `E{2,4}` : signifie de 2 à 4 fois la lettre E majuscule ;
- `E{,5}` : signifie de 0 à 5 fois la lettre E majuscule ;
- `E{8,}` : signifie 8 fois minimum la lettre E majuscule.

Les classes de caractères

Vous pouvez préciser entre crochets plusieurs caractères ou classes de caractères. Par exemple, si vous écrivez `[abcd]`, cela signifie : l'une des lettres parmi `a`, `b`, `c` et `d`.

Pour exprimer des classes, vous pouvez utiliser le tiret `-` entre deux lettres. Par exemple, l'expression `[A-Z]` signifie « une lettre majuscule ». Vous pouvez préciser plusieurs classes ou possibilités dans votre expression. Ainsi, l'expression `[A-Za-z0-9]` signifie « une lettre, majuscule ou minuscule, ou un chiffre ».

Vous pouvez aussi contrôler l'occurence des classes comme nous l'avons vu juste au-dessus. Si vous voulez par exemple rechercher 5 lettres majuscules qui se suivent dans une chaîne, votre expression sera `[A-Z]{5}`.

Les groupes

Je vous donne beaucoup de choses à retenir et vous n'avez pas encore l'occasion de pratiquer. C'est le dernier point sur lequel je vais m'attarder et il sera rapide : comme je l'ai dit plus haut, si vous voulez par exemple contrôler le nombre d'occurences d'un caractère, vous ajoutez derrière un signe particulier (un astérisque, un point d'interrogation, des accolades. . .). Mais si vous voulez appliquer ce contrôle d'occurence à plusieurs caractères, vous allez placer ces caractères entre parenthèses.

```
(cha){2,5}
```

Cette expression sera vérifiée pour les chaînes contenant la séquence `'cha'` répétée entre deux et cinq fois. Les séquences `'cha'` doivent se suivre naturellement.

Les groupes sont également utiles pour remplacer des portions de notre chaîne mais nous y reviendront plus tard, quand sera venue l'heure de la pratique. . . Quoi ? C'est l'heure ? Ah bah c'est parti, alors !

Le module re

Le module `re` a été spécialement conçu pour travailler avec les expressions régulières (*Regular Expressions*). Il définit plusieurs fonctions utiles, que nous allons découvrir, ainsi que des objets propres pour modéliser des expressions.

Chercher dans une chaîne

Nous allons pour ce faire utiliser la fonction `search` du module `re`. Bien entendu, pour pouvoir l'utiliser, il faut l'importer.

```
>>> import re
>>>
```

La fonction `search` attend deux paramètres obligatoires : l'expression régulière, sous la forme d'une chaîne, et la chaîne de caractères dans laquelle on recherche cette expression. Si l'expression est trouvée, la fonction renvoie un objet symbolisant l'expression recherchée. Sinon, elle renvoie `None`.



Certains caractères spéciaux dans nos expressions régulières sont modélisés par l'anti-slash `\`. Vous savez sans doute que Python représente d'autres caractères avec ce symbole. Si vous écrivez dans une chaîne `\n`, Python effectuera un saut de ligne !

Pour symboliser les caractères spéciaux dans les expressions régulières, il est nécessaire d'échapper l'anti-slash en le faisant précéder d'un autre anti-slash. Cela veut dire que pour écrire le caractère spécial `\w`, vous allez devoir écrire `\\w`.

C'est assez peu pratique et parfois gênant pour la lisibilité. C'est pourquoi je vous conseille d'utiliser un format de chaîne que nous n'avons pas vu jusqu'à présent : en plaçant un `r` avant le délimiteur qui ouvre notre chaîne, tous les caractères anti-slash qu'elle contient sont échappés.

```
>>> r'\n'
'\\n'
>>>
```

Si vous avez du mal à voir l'intérêt, je vous conseille simplement de vous rappeler de mettre un `r` avant d'écrire des chaînes contenant des expressions, comme vous allez le voir dans les exemples que je vais vous donner.

Mais revenons à notre fonction `search`. Nous allons mettre en pratique ce que nous avons vu précédemment :

```
>>> re.search(r"abc", "abcdef")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"abc", "abacadaeaf")
```

```
>>> re.search(r"abc*", "ab")
<_sre.SRE_Match object at 0x00AC1800>
>>> re.search(r"abc*", "abccc")
<_sre.SRE_Match object at 0x00AC1640>
>>> re.search(r"chat*", "chateau")
<_sre.SRE_Match object at 0x00AC1800>
>>>
```

Comme vous le voyez, si l'expression est trouvée dans la chaîne, un objet de la classe `_sre.SRE_Match` est renvoyé. Si l'expression n'est pas trouvée, la fonction renvoie `None`.

Cela fait qu'il est extrêmement facile de savoir si une expression est contenue dans une chaîne :

```
if re.match(expression, chaine) is not None:
    # Si l'expression est dans la chaîne
    # Ou alors, plus intuitivement
if re.match(expression, chaine):
```

N'hésitez pas à tester des syntaxes plus complexes et plus utiles. Tenez, par exemple, comment obliger l'utilisateur à saisir un numéro de téléphone ?

Avec le bref descriptif que je vous ai donné dans ce chapitre, vous pouvez théoriquement y arriver. Mais c'est quand même une regex assez complexe alors je vous la donne : prenez le temps de la décortiquer si vous le souhaitez.

Notre regex doit vérifier qu'une chaîne est un numéro de téléphone. L'utilisateur peut saisir un numéro de différentes façons :

- `0X XX XX XX XX`
- `0X-XX-XX-XX-XX`
- `0X.XX.XX.XX.XX`
- `0XXXXXXXXX`

Autrement dit :

- le premier chiffre doit être un 0 ;
- le second chiffre, ainsi que tous ceux qui suivent (9 en tout, sans compter le 0 d'origine) doivent être compris entre 0 et 9 ;
- tous les deux chiffres, on peut avoir un délimiteur optionnel (un tiret, un point ou un espace).

Voici la regex que je vous propose :

```
^0[0-9]([ .-]?[0-9]{2}){4}$
```



ARGH ! C'est illisible ton truc !

Je reconnais que c'est assez peu clair. Décomposons la formule :

- D'abord, on trouve un caractère accent circonflexe ^ qui veut dire qu'on cherche

l'expression au début de la chaîne. Vous pouvez aussi voir, à la fin de la regex, le symbole `$` qui veut dire que l'expression doit être à la fin de la chaîne. Si l'expression doit être au début et à la fin de la chaîne, cela signifie que la chaîne dans laquelle on recherche ne doit rien contenir d'autre que l'expression.
- Nous avons ensuite le `0` qui veut simplement dire que le premier caractère de notre chaîne doit être un `0`.
- Nous avons ensuite une classe de caractère `[0-9]`. Cela signifie qu'après le `0`, on doit trouver un chiffre compris entre 0 et 9 (peut-être 0, peut-être 1, peut-être 2...).
- Ensuite, cela se complique. Vous avez une parenthèse qui matérialise le début d'un groupe. Dans ce groupe, nous trouvons, dans l'ordre :
  - D'abord une classe `[ .-]` qui veut dire « soit un espace, soit un point, soit un tiret ». Juste après cette classe, vous avez un signe `?` qui signifie que cette classe est optionnelle.
  - Après la définition de notre délimiteur, nous trouvons une classe `[0-9]` qui signifie encore une fois « un chiffre entre 0 et 9 ». Après cette classe, entre accolades, vous pouvez voir le nombre de chiffres attendus (2).
- Ce groupe, contenant un séparateur optionnel et deux chiffres, doit se retrouver quatre fois dans notre expression (après la parenthèse fermante, vous trouvez entre accolades le contrôle du nombre d'occurences).

Si vous regardez bien nos numéros de téléphone, vous vous rendez compte que notre regex s'applique aux différents cas présentés. La définition de notre numéro de téléphone n'est pas vraie pour tous les numéros. Cette regex est un exemple et même une base pour vous permettre de saisir le concept.

Si vous voulez que l'utilisateur saisisse un numéro de téléphone, voici le code auquel vous pourriez arriver :

```
import re
chaine = ""
expression = r"^0[0-9]([ .-]?[0-9]{2}){4}$"
while re.search(expression, chaine) is None:
    chaine = input("Saisissez un numéro de téléphone (valide) :
        ")
```

Remplacer une expression

Le remplacement est un peu plus complexe. Je ne vais pas vous montrer d'exemples réellement utiles car ils s'appuient en général sur des expressions assez difficiles à comprendre.

Pour remplacer une partie d'une chaîne de caractères sur la base d'une regex, nous allons utiliser la fonction `sub` du module `re`.

Elle prend trois paramètres :

- l'expression à rechercher ;
- par quoi remplacer cette expression ;
- la chaîne d'origine.

Elle renvoie la chaîne modifiée.

Des groupes numérotés

Pour remplacer une partie de l'expression, on doit d'abord utiliser des groupes. Si vous vous rappelez, les groupes sont indiqués entre parenthèses.

```
(a)b(cd)
```

Dans cet exemple, (a) est le premier groupe et (cd) est le second.

L'ordre des groupes est important dans cet exemple. Dans notre expression de remplacement, nous pouvons appeler nos groupes grâce à \<numéro du groupe>. Pour une fois, on compte à partir de 1.

Ce n'est pas très clair ? Regardez cet exemple simple :

```
>>> re.sub(r"(ab)", r" \1 ", "abcdef")
' ab cdef'
>>>
```

On se contente ici de remplacer 'ab' par ' ab '.

Je vous l'accorde, on serait parvenu au même résultat en utilisant la méthode replace de notre chaîne. Mais les expressions régulières sont bien plus précises que cela : vous commencez à vous en rendre compte, je pense.

Je vous laisse le soin de creuser la question, je préfère ne pas vous présenter tout de suite des expressions trop complexes.

Donner des noms à nos groupes

Nous pouvons également donner des noms à nos groupes. Cela peut être plus clair que de compter sur des numéros. Pour cela, il faut faire suivre la parenthèse ouvrant le groupe d'un point d'interrogation, d'un P majuscule et du nom du groupe entre chevrons <>.

```
(?P<id>[0-9]{2})
```

Dans l'expression de remplacement, on utilisera l'expression \g<nom du groupe> pour symboliser le groupe. Prenons un exemple :

```
>>> texte = """
... nom='Task1', id=8
... nom='Task2', id=31
... nom='Task3', id=127"""
... ...
...
>>> print(re.sub(r"id=(?P<id>[0-9]+)", r"id[\g<id>]", texte))
nom='Task1', id[8]
nom='Task2', id[31]
```

```
nom='Task3', id[127]
...
>>>
```

Des expressions compilées

Si, dans votre programme, vous utilisez plusieurs fois les mêmes expressions régulières, il peut être utile de les compiler. Le module `re` propose en effet de conserver votre expression régulière sous la forme d'un objet que vous pouvez stocker dans votre programme. Si vous devez chercher cette expression dans une chaîne, vous passez par des méthodes de l'expression. Cela vous fait gagner en performances si vous faites souvent appel à cette expression.

Par exemple, j'ai une expression qui est appelée quand l'utilisateur saisit son mot de passe. Je veux vérifier que son mot de passe fait bien six caractères au minimum et qu'il ne contient que des lettres majuscules, minuscules et des chiffres. Voici l'expression à laquelle j'arrive :

```
^[A-Za-z0-9]{6,}$
```

À chaque fois qu'un utilisateur saisit un mot de passe, le programme va appeler `re.search` pour vérifier que celui-ci respecte bien les critères de l'expression. Il serait plus judicieux de conserver l'expression en mémoire.

On utilise pour ce faire la méthode `compile` du module `re`. On stocke la valeur renvoyée (une expression régulière compilée) dans une variable, c'est un objet standard pour le reste.

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
```

Ensuite, vous pouvez utiliser directement cette expression compilée. Elle possède plusieurs méthodes utiles, dont `search` et `sub` que nous avons vu plus haut. À la différence des fonctions du module `re` portant les mêmes noms, elles ne prennent pas en premier paramètre l'expression (celle-ci se trouve directement dans l'objet).

Voyez plutôt :

```
chn_mdp = r"^[A-Za-z0-9]{6,}$"
exp_mdp = re.compile(chn_mdp)
mot_de_passe = ""
while exp_mdp.search(mot_de_passe) is None:
    mot_de_passe = input("Tapez votre mot de passe : ")
```

En résumé

– Les expressions régulières permettent de chercher et remplacer certaines expressions dans des chaînes de caractères.

- Le module `re` de Python permet de manipuler des expressions régulières en Python.
- La fonction `search` du module `re` permet de chercher une expression dans une chaîne.
- Pour remplacer une certaine expression dans une chaîne, on utilise la fonction `sub` du module `re`.
- On peut également compiler les expressions régulières grâce à la fonction `compile` du module `re`.

Chapitre 26

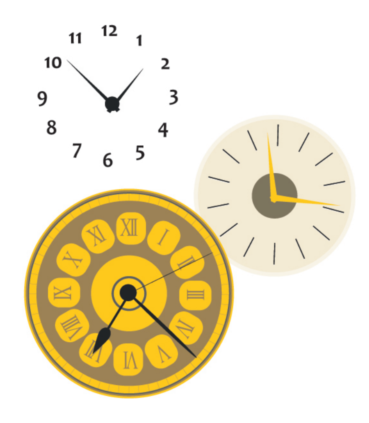
Le temps

Difficulté : 

Exprimer un temps en informatique, cela soulève quelques questions. Disposer d'une mesure du temps dans un programme peut avoir des applications variées : connaître la date et l'heure actuelles et faire remonter une erreur, calculer depuis combien de temps le programme a été lancé, gérer des alarmes programmées, faire des tests de performance. . . et j'en passe !

Il existe plusieurs façons de représenter des temps, que nous allons découvrir maintenant.

Pour bien suivre ce chapitre, vous aurez besoin de maîtriser l'objet : savoir ce qu'est un objet et comment en créer un.



Le module `time`

Le module `time` est sans doute le premier à être utilisé quand on souhaite manipuler des temps de façon simple.

Notez que, dans la documentation de la bibliothèque standard, ce module est classé dans la rubrique **Generic Operating System Services**[1]. Ce n'est pas un hasard : `time` est un module très proche du système. Cela signifie que certaines fonctions de ce module pourront avoir des résultats différents sur des systèmes différents. Pour ma part, je vais surtout m'attarder sur les fonctionnalités les plus génériques possibles afin de ne perdre personne.

Je vous invite à consulter les codes web suivants pour plus de documentation :

- ▷ Bibliothèque standard
  Code web : 110068
- ▷ Module time
  Code web : 299481

Représenter une date et une heure dans un nombre unique

Comment représenter un temps ? Il existe, naturellement, plusieurs réponses à cette question. Celle que nous allons voir ici est sans doute la moins compréhensible pour un humain, mais la plus adaptée à un ordinateur : on stocke la date et l'heure dans un seul entier.



Comment représenter une date et une heure dans un unique entier ?

L'idée retenue a été de représenter une date et une heure en fonction du nombre de secondes écoulées depuis une date précise. La plupart du temps, cette date est l'`Epoch Unix`, le 1er janvier 1970 à 00:00:00.



Pourquoi cette date plutôt qu'une autre ?

Il fallait bien choisir une date de début. L'année 1970 a été considérée comme un bon départ, compte tenu de l'essor qu'a pris l'informatique à partir de cette époque. D'autre part, un ordinateur est inévitablement limité quand il traite des entiers ; dans les langages de l'époque, il fallait tenir compte de ce fait tout simple : on ne pouvait pas compter un nombre de secondes trop important. La date de l'**Epoch** ne pouvait donc pas être trop reculée dans le temps.

Nous allons voir dans un premier temps comment afficher ce fameux nombre de secondes

1. C'est-à-dire les services communs aux différents systèmes d'exploitation.

écoulées depuis le 1er janvier 1970 à 00 :00 :00. On utilise la fonction `time` du module `time`.

```
>>> import time
>>> time.time()
1297642146.562
>>>
```

Cela fait beaucoup ! D'un autre côté, songez quand même que cela représente le nombre de secondes écoulées depuis plus de quarante ans à présent.

Maintenant, je vous l'accorde, ce nombre n'est pas très compréhensible pour un humain. Par contre, pour un ordinateur, c'est l'idéal : les durées calculées en nombre de secondes sont faciles à additionner, soustraire, multiplier... bref, l'ordinateur se débrouille bien mieux avec ce nombre de secondes, ce **timestamp** comme on l'appelle généralement.

Faites un petit test : stockez la valeur renvoyée par `time.time()` dans une première variable, puis quelques secondes plus tard stockez la nouvelle valeur renvoyée par `time.time()` dans une autre variable. Comparez-les, soustrayez-les, vous verrez que cela se fait tout seul :

```
>>> debut = time.time()
>>> # On attend quelques secondes avant de taper la commande
    suivante
... fin = time.time()
>>> print(debut, fin)
1297642195.45 1297642202.27
>>> debut < fin
True
>>> fin - debut # Combien de secondes entre debut et fin ?
6.812000036239624
>>>
```

Vous pouvez remarquer que la valeur renvoyée par `time.time()` n'est pas un entier mais bien un flottant. Le temps ainsi donné est plus précis qu'à une seconde près. Pour des calculs de performance, ce n'est en général pas cette fonction que l'on utilise. Mais c'est bien suffisant la plupart du temps.

La date et l'heure de façon plus présentable

Vous allez me dire que c'est bien joli d'avoir tous nos temps réduits à des nombres mais que ce n'est pas très lisible pour nous. Nous allons découvrir tout au long de ce chapitre des moyens d'afficher nos temps de façon plus élégante et d'obtenir les diverses informations relatives à une date et une heure. Je vous propose ici un premier moyen : une sortie sous la forme d'un objet contenant déjà beaucoup d'informations.

Nous allons utiliser la fonction `localtime` du module `time`.

```
time.localtime()
```

Elle renvoie un objet contenant, dans l'ordre :

1. `tm_year` : l'année sous la forme d'un entier ;
2. `tm_mon` : le numéro du mois (entre 1 et 12) ;
3. `tm_mday` : le numéro du jour du mois (entre 1 et 31, variant d'un mois et d'une année à l'autre) ;
4. `tm_hour` : l'heure du jour (entre 0 et 23) ;
5. `tm_min` : le nombre de minutes (entre 0 et 59) ;
6. `tm_sec` : le nombre de secondes (entre 0 et 61, même si on n'utilisera ici que les valeurs de 0 à 59, c'est bien suffisant) ;
7. `tm_wday` : un entier représentant le jour de la semaine (entre 0 et 6, 0 correspond par défaut au lundi) ;
8. `tm_yday` : le jour de l'année, entre 1 et 366 ;
9. `tm_isdst` : un entier représentant le changement d'heure local.

Comme toujours, si vous voulez en apprendre plus, je vous renvoie à la documentation officielle du module `time`.

Comme je l'ai dit plus haut, nous allons utiliser la fonction `localtime`. Elle prend un paramètre optionnel : le timestamp tel que nous l'avons découvert plus haut. Si ce paramètre n'est pas précisé, `localtime` utilisera automatiquement `time.time()` et renverra donc la date et l'heure actuelles.

```
>>> time.localtime()
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=3,
    tm_min=22, tm_sec=7, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> time.localtime(debut)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
    tm_min=9, tm_sec=55, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> time.localtime(fin)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
    tm_min=10, tm_sec=2, tm_wday=0, tm_yday=45, tm_isdst=0)
>>>
```

Pour savoir à quoi correspond chaque attribut de l'objet, je vous renvoie un peu plus haut. Pour l'essentiel, c'est assez clair je pense. Malgré tout, la date et l'heure renvoyées ne sont pas des plus lisibles. L'avantage de les avoir sous cette forme, c'est qu'on peut facilement extraire une information si on a juste besoin, par exemple, de l'année et du numéro du jour.

Récupérer un timestamp depuis une date

Je vais passer plus vite sur cette fonction car, selon toute vraisemblance, vous l'utiliserez moins souvent. L'idée est, à partir d'une structure représentant les date et heure telles que renvoyées par `localtime`, de récupérer le timestamp correspondant. On utilise pour ce faire la fonction `mktime`.

```
>>> print(debut)
1297642195.45
>>> temps = time.localtime(debut)
>>> print(temps)
time.struct_time(tm_year=2011, tm_mon=2, tm_mday=14, tm_hour=1,
    tm_min=9, tm_sec=55, tm_wday=0, tm_yday=45, tm_isdst=0)
>>> ts_debut = time.mktime(temps)
>>> print(ts_debut)
1297642195.0
>>>
```

Mettre en pause l'exécution du programme pendant un temps déterminé

C'est également une fonctionnalité intéressante, même si vous n'en voyez sans doute pas l'utilité de prime abord. La fonction qui nous intéresse est `sleep` et elle prend en paramètre un nombre de secondes qui peut être sous la forme d'un entier ou d'un flottant. Pour vous rendre compte de l'effet, je vous encourage à tester par vous-mêmes :

```
>>> time.sleep(3.5) # Faire une pause pendant 3,5 secondes
>>>
```

Comme vous pouvez le voir, Python se met en pause et vous devez attendre 3,5 secondes avant que les trois chevrons s'affichent à nouveau.

Formater un temps

Intéressons nous maintenant à la fonction `strftime`. Elle permet de formater une date et heure en la représentant dans une chaîne de caractères.

Elle prend deux paramètres :

- La chaîne de formatage (nous verrons plus bas comment la former).
- Un temps optionnel tel que le renvoie `localtime`. Si le temps n'est pas précisé, c'est la date et l'heure courantes qui sont utilisées par défaut.

Pour construire notre chaîne de formatage, nous allons utiliser plusieurs caractères spéciaux. Python va remplacer ces caractères par leur valeur (la valeur du temps passé en second paramètre ou du temps actuel sinon).

Exemple :

```
time.strftime('%Y')
```

Voici un tableau récapitulatif des quelques symboles que vous pouvez utiliser dans cette chaîne :

Donc pour afficher la date telle qu'on y est habitué en France :

Symbole	Signification
%A	Nom du jour de la semaine
%B	Nom du mois
%d	Jour du mois (de 01 à 31)
%H	Heure (de 00 à 23)
%M	Minute (entre 00 et 59)
%S	Seconde (de 00 à 59)
%Y	Année

```
time.strftime("%A %d %B %Y %H:%M:%S")
```



Mais... c'est en anglais !

Eh oui. Mais avec ce que vous savez déjà et ce que vous allez voir par la suite, vous n'aurez pas de difficulté à personnaliser tout cela !

Bien d'autres fonctions

Le module `time` propose bien d'autres fonctions. Je ne vous ai montré que celles que j'utilise le plus souvent tout en vous présentant quelques concepts du temps utilisé en informatique. Si vous voulez aller plus loin, vous savez quoi faire... non ? Allez, je vous y encourage fortement donc je vous remets le lien vers la documentation du module `time` :

▷ Module time
Code web : 299481

Le module `datetime`

Le module `datetime` propose plusieurs classes pour représenter des dates et heures. Vous n'allez rien découvrir d'absolument spectaculaire dans cette section mais nous nous avançons petit à petit vers une façon de gérer les dates et heures qui est davantage orientée objet.

Encore et toujours, je ne prétends pas remplacer la documentation. Je me contente d'extraire de celle-ci les informations qui me semblent les plus importantes. Je vous encourage, là encore, à jeter un coup d'œil du côté de la documentation du module, que vous trouverez à l'adresse :

▷ Module datetime
Code web : 435379

Représenter une date

Vous le reconnaîtrez probablement avec moi, c'est bien d'avoir accès au temps actuel avec une précision d'une seconde sinon plus... mais parfois, cette précision est inutile. Dans certains cas, on a juste besoin d'une date, c'est-à-dire un jour, un mois et une année. Il est naturellement possible d'extraire cette information de notre timestamp. Le module `datetime` propose une classe `date`, représentant une date, rien qu'une date.

L'objet possède trois attributs :

- `year` : l'année ;
- `month` : le mois ;
- `day` : le jour du mois.



Comment fait-on pour construire notre objet `date` ?

Il y a plusieurs façons de procéder. Le constructeur de cette classe prend trois arguments qui sont, dans l'ordre, l'année, le mois et le jour du mois.

```
>>> import datetime
>>> date = datetime.date(2010, 12, 25)
>>> print(date)
2010-12-25
>>>
```

Il existe deux méthodes de classe qui peuvent vous intéresser :

- `date.today()` : renvoie la date d'aujourd'hui ;
- `date.fromtimestamp(timestamp)` : renvoie la date correspondant au timestamp passé en argument.

Voyons en pratique :

```
>>> import time
>>> import datetime
>>> aujourdhui = datetime.date.today()
>>> aujourdhui
datetime.date(2011, 2, 14)
>>> datetime.date.fromtimestamp(time.time()) # Équivalent à
    date.today
datetime.date(2011, 2, 14)
>>>
```

Et bien entendu, vous pouvez manipuler ces dates simplement et les comparer grâce aux opérateurs usuels, je vous laisse essayer !

Représenter une heure

C'est moins courant mais on peut également être amené à manipuler une heure, indépendemment de toute date. La classe `time` du module `datetime` est là pour cela.

On construit une heure avec non pas trois mais cinq paramètres, tous optionnels :

- `hour` (0 par défaut) : les heures, valeur comprise entre 0 et 23 ;
- `minute` (0 par défaut) : les minutes, valeur comprise entre 0 et 59 ;
- `second` (0 par défaut) : les secondes, valeur comprise entre 0 et 59 ;
- `microsecond` (0 par défaut) : la précision de l'heure en micro-secondes, entre 0 et 1.000.000 ;
- `tzinfo` (`None` par défaut) : l'information de fuseau horaire (je ne détaillerai pas cette information ici).

Cette classe est moins utilisée que `datetime.date` mais elle peut se révéler utile dans certains cas. Je vous laisse faire quelques tests, n'oubliez pas de vous reporter à la documentation du module `datetime` pour plus d'informations.

Représenter des dates et heures

Et nous y voilà ! Vous n'allez pas être bien surpris par ce que nous allons aborder. Nous avons vu une manière de représenter une date, une manière de représenter une heure, mais on peut naturellement représenter une date et une heure dans le même objet, ce sera probablement la classe que vous utiliserez le plus souvent. Celle qui nous intéresse s'appelle `datetime`, comme son module.

Elle prend d'abord les paramètres de `datetime.date` (année, mois, jour) et ensuite les paramètres de `datetime.time` (heures, minutes, secondes, micro-secondes et fuseau horaire).

Voyons dès à présent les deux méthodes de classe que vous utiliserez le plus souvent :

- `datetime.now()` : renvoie l'objet `datetime` avec la date et l'heure actuelles ;
- `datetime.fromtimestamp(timestamp)` : renvoie la date et l'heure d'un timestamp précis.

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2011, 2, 14, 5, 8, 22, 359000)
>>>
```

Il y a bien d'autres choses à voir dans ce module `datetime`. Si vous êtes curieux ou que vous avez des besoins plus spécifiques, que je n'aborde pas ici, référez-vous à la documentation officielle du module.

En résumé

- Le module `time` permet, entre autres, d'obtenir la date et l'heure de votre système.

- La fonction `time` du module `time` renvoie le timestamp actuel.
- La méthode `localtime` du module `time` renvoie un objet isolant les informations d'un timestamp (la date et l'heure).
- Le module `datetime` permet de représenter des dates et heures.
- Les classes `date`, `time` et `datetime` permettent respectivement de représenter des dates, des heures, ainsi que des ensembles « date et heure ».

Chapitre 27

Un peu de programmation système

Difficulté : 

Dans ce chapitre, nous allons découvrir plusieurs modules et fonctionnalités utiles pour interagir avec le système. Python peut servir à créer bien des choses, des jeux, des interfaces, mais il peut aussi faire des scripts systèmes et, dans ce chapitre, nous allons voir comment.

Les concepts que je vais présenter ici risquent d'être plus familiers aux utilisateurs de Linux. Toutefois, pas de panique si vous êtes sur Windows : je vais prendre le temps de vous expliquer à chaque fois tout le nécessaire.



Les flux standard

Pour commencer, nous allons voir comment accéder aux flux standard (entrée standard et sortie standard) et de quelle façon nous devons les manipuler.



À quoi cela ressemble-t-il ?

Vous vous êtes sûrement habitués, quand vous utilisez la fonction `print`, à ce qu'un message s'affiche sur votre écran. Je pense que cela vous paraît même assez logique à présent.

Sauf que, comme pour la plupart de nos manipulations en informatique, le mécanisme qui se cache derrière nos fonctions est plus complexe et puissant qu'il y paraît. Sachez que vous pourriez très bien faire en sorte qu'en utilisant `print`, le texte s'écrive dans un fichier plutôt qu'à l'écran.



Quel intérêt ? `print` est fait pour afficher à l'écran non ?

Pas seulement, non. Mais nous verrons cela un peu plus loin. Pour l'instant, voilà ce que l'on peut dire : quand vous appelez la fonction `print`, si le message s'affiche à l'écran, c'est parce que la sortie standard de votre programme est redirigée vers votre écran.

On distingue trois flux standard :
- L'**entrée standard** : elle est appelée quand vous utilisez `input`. C'est elle qui est utilisée pour demander des informations à l'utilisateur. Par défaut, l'entrée standard est votre clavier.
- La **sortie standard** : comme on l'a vu, c'est elle qui est utilisée pour afficher des messages. Par défaut, elle redirige vers l'écran.
- L'**erreur standard** : elle est notamment utilisée quand Python vous affiche le `traceback` d'une exception. Par défaut, elle redirige également vers votre écran.

Accéder aux flux standard

On peut accéder aux objets représentant ces flux standard grâce au module `sys` qui propose plusieurs fonctions et variables permettant d'interagir avec le système. Nous en reparlerons un peu plus loin dans ce chapitre, d'ailleurs.

```
>>> import sys
>>> sys.stdin # L'entrée standard (standard input)
<_io.TextIOWrapper name='<stdin>' encoding='cp850'>
>>> sys.stdout # La sortie standard (standard output)
<_io.TextIOWrapper name='<stdout>' encoding='cp850'>
```

```
>>> sys.stderr # L'erreur standard (standard error)
<_io.TextIOWrapper name='<stderr>' encoding='cp850'>
>>>
```

Ces objets ne vous rappellent rien ? Vraiment ?

Ils sont de la même classe que les fichiers ouverts grâce à la fonction `open`. Et il n'y a aucun hasard derrière cela.

En effet, pour lire ou écrire dans les flux standard, on utilise les méthodes `read` et `write`.

Naturellement, l'entrée standard `stdin` peut lire (méthode `read`) et les deux sorties `stdout` et `stderr` peuvent écrire (méthode `write`).

Essayons quelque chose :

```
>>> sys.stdout.write("un test")
un test7
>>>
```

Pas trop de surprise, sauf que ce serait mieux avec un saut de ligne à la fin. Là, ce que renvoie la méthode (le nombre de caractères écrits) est affiché juste après notre message.

```
>>> sys.stdout.write("Un test\n")
Un test
8
>>>
```

Modifier les flux standard

Vous pouvez modifier `sys.stdin`, `sys.stdout` et `sys.stderr`. Faisons un premier test :

```
>>> fichier = open('sortie.txt', 'w')
>>> sys.stdout = fichier
>>> print("Quelque chose...")
>>>
```

Ici, rien ne s'affiche à l'écran. En revanche, si vous ouvrez le fichier `sortie.txt`, vous verrez le message que vous avez passé à `print`.



Je ne trouve pas le fichier `sortie.txt`, où est-il ?

Il doit se trouver dans le répertoire courant de Python. Pour connaître l'emplacement de ce répertoire, utilisez le module `os` et la fonction `getcwd`[1].

1. *Get Current Working Directory*

Une petite subtilité : si vous essayez de faire appel à `getcwd` directement, le résultat ne va pas s'afficher à l'écran… il va être écrit dans le fichier. Pour rétablir l'ancienne sortie standard, tapez la ligne :

```
sys.stdout = sys.__stdout__
```

Vous pouvez ensuite faire appel à la fonction `getcwd` :

```
import os
os.getcwd()
```

Dans ce répertoire, vous devriez trouver votre fichier `sortie.txt`.

Si vous avez modifié les flux standard et que vous cherchez les objets d'origine, ceux redirigeant vers le clavier (pour l'entrée) et vers l'écran (pour les sorties), vous pouvez les trouver dans `sys.__stdin__`, `sys.__stdout__` et `sys.__stderr__`.

La documentation de Python nous conseille malgré tout de garder de préférence les objets d'origine sous la main plutôt que d'aller les chercher dans `sys.__stdin__`, `sys.__stdout__` et `sys.__stderr__`.

Voilà qui conclut notre bref aperçu des flux standard. Là encore, si vous ne voyez pas d'application pratique à ce que je viens de vous montrer, cela viendra certainement par la suite.

Les signaux

Les signaux sont un des moyens dont dispose le système pour communiquer avec votre programme. Typiquement, si le système doit arrêter votre programme, il va lui envoyer un signal.

Les signaux peuvent être interceptés dans votre programme. Cela vous permet de déclencher une certaine action si le programme doit se fermer (enregistrer des objets dans des fichiers, fermer les connexions réseau établies avec des clients éventuels, …).

Les signaux sont également utilisés pour faire communiquer des programmes entre eux. Si votre programme est décomposé en plusieurs programmes s'exécutant indépendamment les uns des autres, cela permet de les synchroniser à certains moments clés. Nous ne verrons pas cette dernière fonctionnalité ici, elle mériterait un cours à elle seule tant il y aurait de choses à dire !

Les différents signaux

Le système dispose de plusieurs signaux génériques qu'il peut envoyer aux programmes quand cela est nécessaire. Si vous demandez l'arrêt du programme, un signal particulier lui sera envoyé.

Tous les signaux ne se retrouvent pas sur tous les systèmes d'exploitation, c'est pourquoi je vais surtout m'attacher à un signal : le signal `SIGINT` envoyé à l'arrêt du programme.

Pour plus d'informations, un petit détour par la documentation s'impose, notamment du côté du module `signal`. Vous pouvez y accéder avec le code web suivant :

▷ Module signal
Code web : 699981

Intercepter un `signal`

Commencez par importer le module `signal`.

```
import signal
```

Le signal qui nous intéresse, comme je l'ai dit, se nomme `SIGINT`.

```
>>> signal.SIGINT
2
>>>
```

Pour intercepter ce signal, il va falloir créer une fonction qui sera appelée si le signal est envoyé. Cette fonction prend deux paramètres :

- le signal (plusieurs signaux peuvent être envoyés à la même fonction) ;
- le `frame` qui ne nous intéresse pas ici.

Cette fonction, c'est à vous de la créer. Ensuite, il faudra la connecter avec le signal `SIGINT`.

D'abord, créons notre fonction :

```
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre
        programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)
```



C'est quoi, la dernière ligne ?

On demande simplement à notre programme Python de se fermer. C'est le comportement standard quand on réceptionne un tel signal et notre programme doit bien s'arrêter à un moment ou à un autre.

Pour ce faire, on utilise la fonction `exit` (sortir, en anglais) du module `sys`. Elle prend en paramètre le code de retour du programme.

Pour simplifier, la plupart du temps, si votre programme renvoie `0`, le système comprendra que tout s'est bien passé. Si c'est un entier autre que `0`, le système interprétera cela comme une erreur ayant eu lieu pendant l'exécution de votre programme.

Ici, notre programme s'arrête normalement, on passe donc à `exit 0`.

Connectons à présent notre fonction au signal `SIGINT`, sans quoi notre fonction ne serait jamais appelée.

On utilise pour cela la fonction `signal`. Elle prend en paramètre :

– le signal à intercepter ;
– la fonction que l'on doit connecter à ce signal.

```
signal.signal(signal.SIGINT, fermer_programme)
```

Ne mettez pas les parenthèses à la fin du nom de la fonction. On envoie la référence vers la fonction, on ne l'exécute pas.

Cette ligne va connecter le signal `SIGINT` à la fonction `fermer_programme` que vous avez définie plus haut. Dès que le système enverra ce signal pour fermer le programme, la fonction `fermer_programme` sera appelée.

Pour vérifier que tout fonctionne bien, lancez une boucle infinie dans votre programme :

```
print("Le programme va boucler...")
while True: # Boucle infinie, True est toujours vrai
    continue
```

Je vous remets le code en entier, si cela vous rend les choses plus claires :

```
import signal
import sys

def fermer_programme(signal, frame):
    """Fonction appelée quand vient l'heure de fermer notre
        programme"""
    print("C'est l'heure de la fermeture !")
    sys.exit(0)

# Connexion du signal à notre fonction
signal.signal(signal.SIGINT, fermer_programme)

# Notre programme...
print("Le programme va boucler...")
while True:
    continue
```

Quand vous lancez ce programme, vous voyez un message vous informant que le programme va boucler… et le programme continue de tourner. Il ne s'arrête pas. Il ne fait rien, il boucle simplement mais il va continuer de boucler tant que son exécution n'est pas interrompue.

Dans la fenêtre du programme, tapez CTRL + C sur Windows ou Linux, Cmd + C sur Mac OS X.

Cette combinaison de touches va demander au programme de s'arrêter. Après l'avoir saisie, vous pouvez constater qu'effectivement, votre fonction `fermer_programme` est bien appelée et s'occupe de fermer le programme correctement.

Voilà pour les signaux. Si vous voulez aller plus loin, la documentation est accessible avec le code web indiqué précédemment :

▷ Module signal
Code web : 699981

Interpréter les arguments de la ligne de commande

Python nous offre plusieurs moyens, en fonction de nos besoins, pour interpréter les arguments de la ligne de commande. Pour faire court, ces arguments peuvent être des paramètres que vous passez au lancement de votre programme et qui influeront sur son exécution.

Ceux qui travaillent sur Linux n'auront, je pense, aucun mal à me suivre. Mais je vais faire une petite présentation pour ceux qui viennent de Windows, afin qu'ils puissent suivre sans difficulté.

Si vous êtes allergiques à la console, passez à la suite.

Accéder à la console de Windows

Il existe plusieurs moyens d'accéder à la console de Windows. Celui que j'utilise et que je vais vous montrer passe par le Menu Démarrer.

Ouvrez le Menu Démarrer et cliquez sur `exécuter...`. Dans la fenêtre qui s'ouvre, tapez `cmd` puis appuyez sur `Entrée`.

Vous devriez vous retrouver dans une fenêtre en console, vous donnant plusieurs informations propres au système.

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\utilisateur>
```

Ce qui nous intéresse, c'est la dernière ligne. C'est un chemin qui vous indique à quel endroit de l'arborescence vous vous trouvez. Il y a toutes les chances que ce chemin soit le répertoire utilisateur de votre compte.

```
C:\Documents and Settings\utilisateur>
```

Nous allons commencer par nous déplacer dans le répertoire contenant l'interpréteur Python. Là encore, si vous n'avez rien changé lors de l'installation de Python, le chemin correspondant est `C:\pythonXY`, `XY` représentant les deux premiers chiffres de votre version de Python. Avec Python 3.2, ce sera donc probablement `C:\python32`.

Déplacez-vous dans ce répertoire grâce à la commande `cd`.

```
C:\Documents and Settings\utilisateur>cd C:\python32
```

```
C:\Python32>
```

Si tout se passe bien, la dernière ligne vous indique que vous êtes bien dans le répertoire Python.

En vérité, vous pouvez appeler Python de n'importe où dans l'arborescence mais ce sera plus simple si nous sommes dans le répertoire de Python pour commencer.

Accéder aux arguments de la ligne de commande

Nous allons une fois encore faire appel à notre module `sys`. Cette fois, nous allons nous intéresser à sa variable `argv`.

Créez un nouveau fichier Python. Sur Windows, prenez bien soin de l'enregistrer dans le répertoire de Python (`C:\python32` sous `Python 3.2`).

Placez-y le code suivant :

```python
import sys
print(sys.argv)
```

`sys.argv` contient une liste des arguments que vous passez en ligne de commande, au moment de lancer le programme. Essayez donc d'appeler votre programme depuis la ligne de commande en lui passant des arguments.

Sur Windows :

```
C:\Python32>python test_console.py
['test_console.py']
C:\Python32>python test_console.py arguments
['test_console.py', 'arguments']
C:\Python32>python test_console.py argument1 argument2 argument3
['test_console.py', 'argument1', 'argument2', 'argument3']
C:\Python32>
```

Comme vous le voyez, le premier élément de `sys.argv` contient le nom du programme, de la façon dont vous l'avez appelé. Le reste de la liste contient vos arguments (s'il y en a).

Note : vous pouvez très bien avoir des arguments contenant des espaces. Dans ce cas, vous devez alors encadrer l'argument de guillemets :

```
C:\Python32>python test_console.py "un argument avec des espaces"
['test_console.py', 'un argument avec des espaces']
C:\Python32>
```

Interpréter les arguments de la ligne de commande

Accéder aux arguments, c'est bien, mais les interpréter peut être utile aussi.

Des actions simples

Parfois, votre programme devra déclencher plusieurs actions en fonction du premier paramètre fourni. Par exemple, en premier argument, vous pourriez préciser l'une des valeurs suivantes : `start` pour démarrer une opération, `stop` pour l'arrêter, `restart` pour la redémarrer, `status` pour connaître son état… bref, les utilisateurs de Linux ont sûrement bien plus d'exemples à l'esprit.

Dans ce cas de figure, il n'est pas vraiment nécessaire d'interpréter les arguments de la ligne de commande, comme on va le voir. Notre programme Python ressemblerait simplement à cela :

```python
import sys

if len(sys.argv) < 2:
    print("Précisez une action en paramètre")
    sys.exit(1)

action = sys.argv[1]

if action == "start":
    print("On démarre l'opération")
elif action == "stop":
    print("On arrête l'opération")
elif action == "restart":
    print("On redémarre l'opération")
elif action == "status":
    print("On affiche l'état (démarré ou arrêté ?) de l'opération")
else:
    print("Je ne connais pas cette action")
```



Il est nécessaire de passer par la ligne de commande pour tester ce programme.

Des options plus complexes

Mais la ligne de commande permet également de transmettre des arguments plus complexes comme des options. La plupart du temps, nos options sont sous la forme : `-option_courte` (une seule lettre), `--option_longue`, suivie d'un argument ou non.

Souvent, une option courte est accessible aussi depuis une option longue.

Ici, mon exemple va être tiré de Linux mais vous n'avez pas vraiment besoin d'être sur Linux pour le comprendre, rassurez-vous.

La commande `ls` permet d'afficher le contenu d'un répertoire. On peut lui passer en paramètres plusieurs options qui influent sur ce que la commande va afficher au final.

Par exemple, pour afficher tous les fichiers (cachés ou non) du répertoire, on utilise l'option courte `a`.

```
$ ls -a
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Cette option courte est accessible depuis une option longue, `all`. Vous arrivez donc au même résultat en tapant :

```
$ ls --all
.  ..  fichier1.txt  .fichier_cache.txt  image.png
$
```

Pour récapituler, nos options courtes sont précédées d'un seul tiret et composées d'une seule lettre. Les options longues sont précédées de deux tirets et composées de plusieurs lettres.

Certaines options attendent un argument, à préciser juste après l'option.

Par exemple (toujours sur Linux), pour afficher les premières lignes d'un fichier, vous pouvez utiliser la commande `head`. Si vous voulez afficher les X premières lignes d'un fichier, vous utiliserez la commande `head -n X`.

```
$ head -n 5 fichier.txt
ligne 1
ligne 2
ligne 3
ligne 4
ligne 5
$
```

Dans ce cas, l'option `-n` attend un argument qui est le nombre de lignes à afficher.

Interpréter ces options grâce à Python

Cette petite présentation faite, revenons à Python.

Nous allons nous intéresser au module `getopt` qui est utile, justement, pour interpréter les arguments de la ligne de commande selon un certain schéma.

La fonction qui nous intéresse s'appelle `getopt`, comme son module. Elle prend en paramètres :

- La liste des arguments à interpréter. Dans notre cas, ce sera toujours `sys.argv[1:]` (on ne prend pas le premier paramètre qui est le nom du programme).

– Une chaîne représentant la suite des options courtes (d'une seule lettre donc).
– Une liste contenant plusieurs chaînes de caractères, représentant les options longues correspondant aux options courtes. Elles doivent être fournies dans le même ordre que les options courtes. Ce paramètre est facultatif : les options courtes peuvent très bien n'avoir aucune option longue correspondante.

Un exemple s'impose je pense :

```
import sys
import getopt
getopt.getopt(sys.argv[1:], "h", ["help"])
```

Cela signifie que notre argument de la ligne de commande peut éventuellement contenir une option, soit `-h` sous sa forme courte, soit `--help` sous sa forme longue.

Si vous voulez que vos options soient suivies de paramètres, faites suivre l'option courte d'un signe deux point `:` et l'option longue correspondante d'un signe égal `=`.

```
getopt.getopt(sys.argv[1:], "n:", ["number="])
```

Si une erreur se produit durant l'interprétation des options (ce qui signifie que l'utilisateur ne les a pas précisées comme elles devaient l'être), alors une exception de type `getopt.GetoptError` est levée. C'est pourquoi on place souvent un appel à `getopt.getopt` dans un bloc `try`.

La fonction renvoie deux informations : la liste des options, sous la forme d'une liste de couples (non verrons cela plus bas), et la liste des arguments non interprétés.

Je pense qu'un exemple avec tout le code d'interprétation sera plus parlant. Il est tiré de la documentation officielle du module `getopt`, accessible grâce au code web suivant :

▷ Module getopt
Code web : 981497

```
import sys
import getopt

try:
    opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "
        output="])
except getopt.GetoptError as err:
    # Affiche l'aide et quitte le programme
    print(err) # Va afficher l'erreur en anglais
    usage() # Fonction à écrire rappelant la syntaxe de la
        commande
    sys.exit(2)

output = None
verbose = False
for o, a in opts:
    if o == "-v":
        # On place l'option 'verbose' à True
        verbose = True
```

```
    elif o in ("-h", "--help"):
        # On affiche l'aide
        usage()
        sys.exit()
    elif o in ("-o", "--output"):
        output = a
    else:
        print("Option {} inconnue".format(o))
        sys.exit(2)
```

Vous pouvez passer à notre programme plusieurs options :

- `-h` ou `--help` : affichera l'aide de notre programme ;
- `-v` : activera l'option `verbose`, en général utilisée pour « faire parler » le programme de façon détaillée ;
- `-o` ou `--output`, cette option attend un paramètre.

Dans cet exemple, on ne traite pas les arguments supplémentaires passés au programme mais, si vous en avez besoin, ils se trouveront dans votre variable `args`.

Si vous voulez en savoir plus, avoir d'autres exemples ou plus d'informations, je vous redirige naturellement vers la documentation du module `getopt` indiquée plus haut.

Si vous avez besoin de quelque chose de plus puissant, sachez qu'il existe, depuis Python 3.2, les modules `argparse` et `optparse`. Je vous invite à les consulter avec les codes web suivants :

▷ Module argparse
Code web : 664756

▷ Module optparse
Code web : 133756

Exécuter une commande système depuis Python

Nous allons ici nous intéresser à la façon d'exécuter des commandes depuis Python. Nous allons voir deux moyens, il en existe cependant d'autres.

Ceux que je vais présenter ont l'avantage de fonctionner sur Windows.

La fonction `system`

Vous vous souvenez peut-être de cette fonction du module `os`. Elle prend en paramètre une commande à exécuter, affiche le résultat de la commande et renvoie son code de retour.

```
os.system("ls") # Sur Linux
os.system("dir") # Sur Windows
```

Vous pouvez capturer le code de retour de la commande mais vous ne pouvez pas capturer le retour affiché par la commande.

En outre, la fonction `system` exécute un environnement particulier rien que pour votre commande. Cela veut dire, entre autres, que `system` retournera tout de suite même si la commande tourne toujours.

En gros, si vous faites `os.system(« sleep 5 »)`, le programme ne s'arrêtera pas pendant cinq secondes.

La fonction `popen`

Cette fonction se trouve également dans le module `os`. Elle prend également en paramètre une commande.

Toutefois, au lieu de renvoyer le code de retour de la commande, elle renvoie un objet, un *pipe*[2] qui vous permet de lire le retour de la commande.

Un exemple sur Linux :

```
>>> import os
>>> cmd = os.popen("ls")
>>> cmd
<os._wrap_close object at 0x7f81d16554d0>
>>> cmd.read()
'fichier1.txt\nimage.png\n'
>>>
```

Le fait de lire le *pipe* bloque le programme jusqu'à ce que la commande ait fini de s'exécuter.

Je vous ai dit qu'il existait d'autres moyens. Et au-delà de cela, vous avez beaucoup d'autres choses intéressantes dans le module `os` vous permettant d'interagir avec le système... et pour cause !

En résumé

- Le module `sys` propose trois objets permettant d'accéder aux flux standard : `stdin`, `stdout` et `stderr`.
- Le module `signal` permet d'intercepter les signaux envoyés à notre programme.
- Le module `getopt` permet d'interpréter les arguments passés en console à notre programme.
- Enfin, le module `os` possède, entre autres, plusieurs fonctions pour envoyer des commandes au système.

2. Mot anglais pour un « tuyau ».

Chapitre 28

Un peu de mathématiques

Difficulté : 

Dans ce chapitre, nous allons découvrir trois modules. Je vous ai déjà fait utiliser certains de ces modules, ce sera ici l'occasion de revenir dessus plus en détail.

- Le module `math` qui propose un bon nombre de fonctions mathématiques.
- Le module `fractions`, dont nous allons surtout voir la classe `Fraction`, permettant... vous l'avez deviné ? De modéliser des fractions.
- Et enfin le module `random` que vous connaissez de par nos TP et que nous allons découvrir plus en détail ici.



Pour commencer, le module math

Le module `math`, vous le connaissez déjà : nous l'avons utilisé comme premier exemple de module créé par Python. Vous avez peut-être eu la curiosité de regarder l'aide du module pour voir quelles fonctions y étaient définies. Dans tous les cas, je fais un petit point sur certaines de ces fonctions.

Je ne vais pas m'attarder très longtemps sur ce module en particulier car il est plus vraisemblable que vous cherchiez une fonction précise et que la documentation sera, dans ce cas, plus accessible et explicite.

Fonctions usuelles

Vous vous souvenez des opérateurs `+`, `-`, `*`, `/` et `%` j'imagine, je ne vais peut-être pas y revenir.

Trois fonctions pour commencer notre petit tour d'horizon :

```
>>> math.pow(5, 2) # 5 au carré
25.0
>>> 5 ** 2 # Pratiquement identique à pow(5, 2)
25
>>> math.sqrt(25) # Racine carrée de 25 (square root)
5.0
>>> math.exp(5) # Exponentielle
148.4131591025766
>>> math.fabs(-3) # Valeur absolue
3.0
>>>
```

Il y a bel et bien une différence entre l'opérateur `**` et la fonction `math.pow`. La fonction renvoie toujours un flottant alors que l'opérateur renvoie un entier quand cela est possible.

Un peu de trigonométrie

Avant de voir les fonctions usuelles en trigonométrie, j'attire votre attention sur le fait que les angles, en Python, sont donnés et renvoyés en radians (rad).

Pour rappel :

$$1 \text{ rad} = 57{,}29 \text{ degrés}$$

Cela étant dit, il existe déjà dans le module `math` les fonctions qui vont nous permettre de convertir simplement nos angles.

```
math.degrees(angle_en_radians) # Convertit en degrés
math.radians(angle_en_degrés) # Convertit en radians
```

Voyons maintenant quelques fonctions. Elles se nomment, sans surprise :

- `cos` : cosinus ;
- `sin` : sinus ;
- `tan` : tangente ;
- `acos` : arc cosinus ;
- `asin` : arc sinus ;
- `atan` : arc tangente.

Arrondir un nombre

Le module `math` nous propose plusieurs fonctions pour arrondir un nombre selon différents critères :

```
>>> math.ceil(2.3) # Renvoie le plus petit entier >= 2.3
3
>>> math.floor(5.8) # Renvoie le plus grand entier <= 5.8
5
>>> math.trunc(9.5) # Tronque 9.5
9
>>>
```

Quant aux constantes du module, elles ne sont pas nombreuses : `math.pi` naturellement, ainsi que `math.e`.

Voilà, ce fut rapide mais suffisant, sauf si vous cherchez quelque chose de précis. En ce cas, un petit tour du côté de la documentation officielle du module `math` s'impose. Consultez-la grâce au code web suivant :

▷ Module math
Code web : 197635

Des fractions avec le module `fractions`

Ce module propose, entre autres, de manipuler des objets modélisant des fractions. C'est la classe `Fraction` du module qui nous intéresse :

```
from fractions import Fraction
```

Créer une fraction

Le constructeur de la classe `Fraction` accepte plusieurs types de paramètres :

- Deux entiers, le numérateur et le dénominateur (par défaut le numérateur vaut `0` et le dénominateur `1`). Si le dénominateur est `0`, une exception `ZeroDivisionError` est levée.
- Une autre fraction.

– Une chaîne sous la forme `'numerateur/denominateur'`.

```
>>> un_demi = Fraction(1, 2)
>>> un_demi
Fraction(1, 2)
>>> un_quart = Fraction('1/4')
>>> un_quart
Fraction(1, 4)
>>> autre_fraction = Fraction(-5, 30)
>>> autre_fraction
Fraction(-1, 6)
>>>
```



Ne peut-on pas créer des fractions depuis un flottant ?

Si, mais pas dans le constructeur. Pour créer une fraction depuis un flottant, on utilise la méthode de classe `from_float` :

```
>>> Fraction.from_float(0.5)
Fraction(1, 2)
>>>
```

Et pour retomber sur un flottant, rien de plus simple :

```
>>> float(un_quart)
0.25
>>>
```

Manipuler les fractions

Maintenant, quel intérêt d'avoir nos nombres sous cette forme ? Surtout pour la précision des calculs. Les fractions que nous venons de voir acceptent naturellement les opérateurs usuels :

```
>>> un_dixieme = Fraction(1, 10)
>>> un_dixieme + un_dixieme + un_dixieme
Fraction(3, 10)
>>>
```

Alors que :

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>>
```

Bien sûr, la différence n'est pas énorme mais elle est là. Tout dépend de vos besoins en termes de précision.

D'autres calculs ?

```
>>> un_dixieme * un_quart
Fraction(1, 40)
>>> un_dixieme + 5
Fraction(51, 10)
>>> un_demi / un_quart
Fraction(2, 1)
>>> un_quart / un_demi
Fraction(1, 2)
>>>
```

Voilà. Cette petite démonstration vous suffira si ce module vous intéresse. Et si elle ne suffit pas, rendez-vous sur la documentation officielle du module fractions accessible avec le code web suivant :

▷ Module fractions
Code web : 320808

Du pseudo-aléatoire avec `random`

Le module `random`, vous l'avez également utilisé pendant nos TP. Nous allons voir quelques fonctions de ce module, le tour d'horizon sera rapide !

Du pseudo-aléatoire

L'ordinateur est une machine puissante, capable de faire beaucoup de choses. Mais lancer les dés n'est pas son fort. Une calculatrice standard n'a aucune difficulté à additionner, soustraire, multiplier ou diviser des nombres. Elle peut même faire des choses bien plus complexes. Mais, pour un ordinateur, choisir un nombre au hasard est bien plus compliqué qu'il n'y paraît.

Ce qu'il faut bien comprendre, c'est que derrière notre appel à `random.randrange` par exemple, Python va faire un véritable calcul pour trouver un nombre aléatoire. De ce fait, le nombre généré n'est pas réellement aléatoire puisqu'un calcul identique, effectué dans les mêmes conditions, donnera le même nombre. Cependant, les algorithmes mis en place pour générer de l'aléatoire sont maintenant suffisamment complexes pour que les nombres générés ressemblent bien à une série aléatoire. Souvenez-vous toutefois que, pour un ordinateur, le véritable hasard ne peut pas exister.

La fonction `random`

Cette fonction, on ne l'utilisera peut-être pas souvent de manière directe mais elle est implicitement utilisée par le module quand on fait appel à `randrange` ou `choice` que

nous verrons plus bas.

Elle génère un nombre pseudo-aléatoire compris entre 0 et 1. Ce sera donc naturellement un flottant :

```
>>> import random
>>> random.random()
0.9565461152605507
>>>
```

`randrange` et `randint`

La fonction `randrange` prend trois paramètres :

- la marge inférieure de l'intervalle ;
- la marge supérieure de l'intervalle ;
- l'écart entre chaque valeur de l'intervalle (`1` par défaut).



Que représente le dernier paramètre ?

Prenons un exemple, ce sera plus simple :

```
random.randrange(5, 10, 2)
```

Cette instruction va chercher à générer un nombre aléatoire entre `5` inclus et `10` non inclus, avec un écart de `2` entre chaque valeur. Elle va donc chercher dans la liste des valeurs `[5, 7, 9]`.

Si vous ne précisez pas de troisième paramètre, il vaudra `1` par défaut (c'est le comportement attendu la plupart du temps).

La fonction `randint` prend deux paramètres :

- là encore, la marge inférieure de l'intervalle ;
- la marge supérieure de l'intervalle, cette fois incluse.

Pour tirer au hasard un nombre entre `1` et `6`, il est donc plus intuitif de faire :

```
random.randint(1, 6)
```

Opérations sur des séquences

Nous allons voir deux fonctions : la première, `choice`, renvoie au hasard un élément d'une séquence passée en paramètre :

```
>>> random.choice(['a', 'b', 'k', 'p', 'i', 'w', 'z'])
'k'
>>>
```

La seconde s'appelle `shuffle`. Elle prend en paramètre une séquence et la mélange ; elle modifie donc la séquence qu'on lui passe et ne renvoie rien :

```
>>> liste = ['a', 'b', 'k', 'p', 'i', 'w', 'z']
>>> random.shuffle(liste)
>>> liste
['p', 'k', 'w', 'z', 'i', 'b', 'a']
>>>
```

Voilà. Là encore, ce fut rapide mais si vous voulez aller plus loin, vous savez ou aller... droit vers la documentation officielle de Python, avec le code web suivant :

▷ Documentation random
Code web : 915051

En résumé

- Le module `math` possède plusieurs fonctions et constantes mathématiques usuelles.
- Le module `fractions` possède le nécessaire pour manipuler des fractions, parfois utiles pour la précision des calculs.
- Le module `random` permet de générer des nombres pseudo-aléatoires.

Chapitre 29

Gestion des mots de passe

Difficulté : 

Dans ce chapitre, nous allons nous intéresser aux mots de passe et à la façon de les gérer en Python, c'est-à-dire de les réceptionner et de les protéger.

Nous allons découvrir deux modules dans ce chapitre : d'abord `getpass` qui permet de demander un mot de passe à l'utilisateur, puis `hashlib` qui permet de chiffrer le mot de passe réceptionné.

****

Réceptionner un mot de passe saisi par l'utilisateur

Vous allez me dire, j'en suis sûr, qu'on a déjà une façon de réceptionner une saisie de l'utilisateur. Cette méthode, on l'a vue assez tôt dans le cours : il s'agit naturellement de la fonction `input`.

Mais `input` n'est pas très discrète. Si vous saisissez un mot de passe confidentiel, il apparaît de manière visible à l'écran, ce qui n'est pas toujours souhaitable. Quand on tape un mot de passe, c'est même rarement souhaité !

C'est ici qu'intervient le module `getpass`. La fonction qui nous intéresse porte le même nom que le module. Elle va réagir comme `input`, attendre une saisie de l'utilisateur et la renvoyer. Mais à la différence d'`input`, elle ne va pas afficher ce que l'utilisateur saisit.

Faisons un essai :

```
>>> from getpass import getpass
>>> mot_de_passe = getpass()
Password:
>>> mot_de_passe
'un mot de passe'
>>>
```

Comme vous le voyez... bah justement on ne voit rien ! Le mot de passe que l'on tape est invisible. Vous appuyez sur les touches de votre clavier mais rien ne s'affiche. Cependant, vous écrivez bel et bien et, quand vous appuyez sur `Entrée`, la fonction `getpass` renvoie ce que vous avez saisi.

Ici, on le stocke dans la variable `mot_de_passe`. C'est plus discret qu'`input`, reconnaissez-le !

Bon, il reste un détail, mineur certes, mais un détail quand même : le `prompt` par défaut, c'est-à-dire le message qui vous invite à saisir votre mot de passe, est en anglais. Heureusement, il s'agit tout simplement d'un paramètre facultatif de la fonction :

```
>>> mot_de_passe = getpass("Tapez votre mot de passe : ")
Tapez votre mot de passe :
>>>
```

C'est mieux.

Bien entendu, tous les mots de passe que vous réceptionnerez ne viendront pas forcément d'une saisie directe d'un utilisateur. Mais, dans ce cas précis, la fonction `getpass` est bien utile. À la fin de ce chapitre, nous verrons une utilisation complète de cette fonction, incluant réception et chiffrement de notre mot de passe en prime, deux en un.

Chiffrer un mot de passe

Cette fois-ci, nous allons nous intéresser au module `hashlib`. Mais avant de vous montrer comment il fonctionne, quelques explications s'imposent.

Chiffrer un mot de passe ?

La première question qu'on pourrait légitimement se poser est « pourquoi protéger un mot de passe ? ». Je suis sûr que vous pouvez trouver par vous-mêmes pas mal de réponses : il est un peu trop facile de récupérer un mot de passe s'il est stocké ou transmis en clair. Et, avec un mot de passe, on peut avoir accès à beaucoup de choses : je n'ai pas besoin de vous l'expliquer. Cela fait que généralement, quand on a besoin de stocker un mot de passe ou de le transmettre, on le chiffre.

Maintenant, qu'est-ce que le chiffrement ? *A priori*, l'idée est assez simple : en partant d'un mot de passe, n'importe lequel, on arrive à une seconde chaîne de caractères, complètement incompréhensible.



Quel intérêt ?

Eh bien, si vous voyez passer, devant vos yeux, une chaîne de caractères comme `b47ea832576a75814e13351dcc97eaa985b9c6b7`, vous ne pouvez pas vraiment deviner le mot de passe qui se cache derrière.

Et l'ordinateur ne peut pas le déchiffrer si facilement que cela non plus. Bien sûr, il existe des méthodes pour déchiffrer un mot de passe mais nous ne les verrons certainement pas ici. Nous, ce que nous voulons savoir, c'est comment protéger nos mots de passe, pas comment déchiffrer ceux des autres !



Comment fonctionne le chiffrement ?

Grave question. D'abord, il existe plusieurs techniques ou **algorithmes** de chiffrement. Chiffrer un mot de passe avec un certain algorithme ne donne pas le même résultat qu'avec un autre algorithme.

Ensuite, l'algorithme, quel qu'il soit, est assez complexe. Je serais bien incapable de vous expliquer en détail comment cela marche, on fait appel à beaucoup de concepts mathématiques relativement poussés.

Mais si vous voulez faire un exercice, je vous propose quelque chose d'amusant qui vous donnera une meilleure idée du chiffrement.

Commencez par numéroter toutes les lettres de l'alphabet (de **a** à **z**) de **1** à **26**. Représentez l'ensemble des valeurs dans un tableau, ce sera plus simple.

A (1)	B (2)	C (3)	D (4)	E (5)	F (6)	
G (7)	H (8)	I (9)	J (10)	K (11)	L (12)	M (13)
N (14)	O (15)	P (16)	Q (17)	R (18)	S (19)	
T (20)	U (21)	V (22)	W (23)	X (24)	Y (25)	Z (26)

Maintenant, supposons que nous allons chercher à chiffrer des prénoms. Pour cela, nous allons baser notre exemple sur un calcul simple : dans le tableau ci-dessus, prenez la valeur numérique de chaque lettre constituant le prénom et additionnez l'ensemble des valeurs obtenues.

Par exemple, partons du prénom *Eric*. Quatre lettres, cela ira vite. Oubliez les accents, les majuscules et minuscules. On a un **E (5)**, un **R (18)**, un **I (9)** et un **C (3)**. En ajoutant les valeurs de chaque lettre, on a donc **5 + 18 + 9 + 3**, ce qui donne **35**.

Conclusion : en chiffrant *Eric* grâce à notre algorithme, on obtient le nombre **35**.

C'est l'idée derrière le chiffrement même si, en réalité, les choses sont beaucoup plus complexes. En outre, au lieu d'avoir un chiffre en sortie, on a généralement plutôt une chaîne de caractères.

Mais prenez cet exemple pour vous amuser, si vous voulez. Appliquez notre algorithme à plusieurs prénoms. Si vous vous sentez d'attaque, essayez de faire une fonction Python qui prenne en paramètre notre chaîne et renvoie un chiffre, ce n'est pas bien difficile.

Vous pouvez maintenant vous rendre compte que derrière un nombre tel que **35**, il est plutôt difficile de deviner que se cache le prénom *Eric* !

Si vous faites le test sur les prénoms *Louis* et *Jacques*, vous vous rendrez compte… qu'ils produisent le même résultat, **76**. En effet :

- Louis = 12 + 15 + 21 + 9 + 19 = 76
- Jacques = 10 + 1 + 3 + 17 + 21 + 5 + 19 = 76

C'est ce qu'on appelle une **collision** : en prenant deux chaînes différentes, on obtient le même chiffrement au final.

Les algorithmes que nous allons voir dans le module `hashlib` essayent de minimiser, autant que possible, les collisions. Celui que nous venons juste de voir en est plein : il suffit de changer de place les lettres de notre prénom et nous retombons sur le même nombre, après tout.

Voilà. Fin de l'exercice, on va se pencher sur le module `hashlib` maintenant.

Chiffrer un mot de passe

On peut commencer par importer le module `hashlib` :

```python
import hashlib
```

On va maintenant choisir un algorithme. Pour nous aider dans notre choix, le module `hashlib` nous propose deux listes :

- `algorithms_guaranteed` : les algorithmes garantis par Python, les mêmes d'une

plateforme à l'autre. Si vous voulez faire des programmes portables, il est préférable d'utiliser un de ces algorithmes :

```
>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}
>>>
```

– `algorithms_available` : les algorithmes disponibles sur votre plateforme. Tous les algorithmes garantis s'y trouvent, plus quelques autres propres à votre système.

Dans ce chapitre, nous allons nous intéresser à `sha1`.

Pour commencer, nous allons créer notre objet `SHA1`. On va utiliser le constructeur `sha1` du module `hashlib`. Il prend en paramètre une chaîne, mais une chaîne de **bytes** (octets).

Pour obtenir une chaîne de **bytes** depuis une chaîne `str`, on peut utiliser la méthode `encode`. Je ne vais pas rentrer dans le détail des encodages ici. Pour écrire directement une chaîne **bytes** sans passer par une chaîne `str`, vous avez une autre possibilité consistant à mettre un `b` minuscule avant l'ouverture de votre chaîne :

```
>>> b'test'
b'test'
>>>
```

Générons notre mot de passe :

```
>>> mot_de_passe = hashlib.sha1(b"mot de passe")
>>> mot_de_passe
<sha1 HASH object @ 0x00BF0ED0>
>>>
```

Pour obtenir le chiffrement associé à cet objet, on a deux possibilités :

– la méthode `digest`, qui renvoie un type **bytes** contenant notre mot de passe chiffré ;
– la méthode `hexdigest`, qui renvoie une chaîne `str` contenant une suite de symboles hexadécimaux (de 0 à 9 et de A à F).

C'est cette dernière méthode que je vais montrer ici, parce qu'elle est préférable pour un stockage en fichier si les fichiers doivent transiter d'une plateforme à l'autre.

```
>>> mot_de_passe.hexdigest()
'b47ea832576a75814e13351dcc97eaa985b9c6b7'
>>>
```



Et pour déchiffrer ce mot de passe ?

On ne le déchiffre pas. Si vous voulez savoir si le mot de passe saisi par l'utilisateur correspond au chiffrement que vous avez conservé, chiffrez le mot de passe qui vient

d'être saisi et comparez les deux chiffrements obtenus :

```python
import hashlib
from getpass import getpass

chaine_mot_de_passe = b"azerty"
mot_de_passe_chiffre = hashlib.sha1(chaine_mot_de_passe).hexdigest()

verrouille = True
while verrouille:
    entre = getpass("Tapez le mot de passe : ") # azerty
    # On encode la saisie pour avoir un type bytes
    entre = entre.encode()

    entre_chiffre = hashlib.sha1(entre).hexdigest()
    if entre_chiffre == mot_de_passe_chiffre:
        verrouille = False
    else:
        print("Mot de passe incorrect")

print("Mot de passe accepté...")
```

Cela me semble assez clair. Nous avons utilisé l'algorithme `sha1`, il en existe d'autres comme vous pouvez le voir dans `hashlib.algorithms_available`.

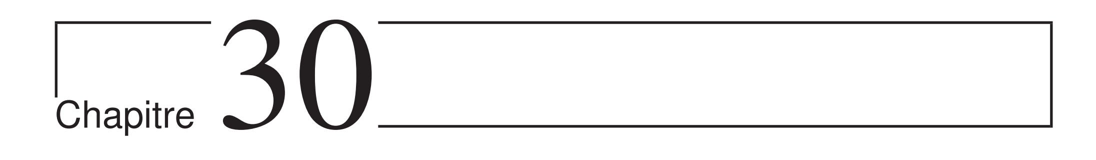
Je m'arrête pour ma part ici ; si vous voulez aller plus loin, je vous redirige vers les codes web suivants :

▷ Module getpass
Code web : 246938

▷ Module hashlib
Code web : 331371

En résumé

- Pour demander à l'utilisateur de saisir un mot de passe, on peut utiliser le module `getpass`.
- La fonction `getpass` du module `getpass` fonctionne de la même façon que `input`, sauf qu'elle n'affiche pas ce que l'utilisateur saisit.
- Pour chiffrer un mot de passe, on va utiliser le module `hashlib`.
- Ce module contient en attributs les différents algorithmes pouvant être utilisés pour chiffrer nos mots de passe.



Chapitre 30

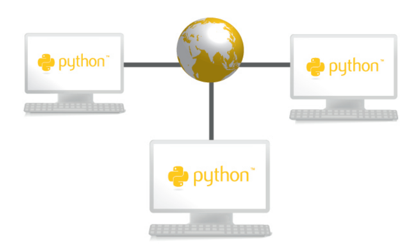
Le réseau

Difficulté : 

Vaste sujet que le réseau ! Si je devais faire une présentation détaillée, ou même parler des réseaux en général, il me faudrait bien plus d'un chapitre rien que pour la théorie.

Dans ce chapitre, nous allons donc apprendre à faire communiquer deux applications grâce aux **sockets**, des objets qui permettent de connecter un client à un serveur et de transmettre des données de l'un à l'autre.

Si cela ne vous met pas l'eau à la bouche. . .



Brève présentation du réseau

Comme je l'ai dit plus haut, le réseau est un sujet bien trop vaste pour que je le présente en un unique chapitre. On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes. Dans ce cas, elles se connectent grâce au réseau local ou à Internet.

Il existe plusieurs protocoles de communication en réseau. Si vous voulez, c'est un peu comme la communication orale : pour que les échanges se passent correctement, les deux (ou plus) parties en présence doivent parler la même langue. Nous allons ici parler du protocole **TCP**.

Le protocole TCP

L'acronyme de ce protocole signifie *Transmission Control Protocol*, soit « protocole de contrôle de transmission ». Concrètement, il permet de connecter deux applications et de leur faire échanger des informations.

Ce protocole est dit « orienté connexion », c'est-à-dire que les applications sont connectées pour communiquer et que l'on peut être sûr, quand on envoie une information au travers du réseau, qu'elle a bien été réceptionnée par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent rétablir la connexion pour communiquer de nouveau.

Cela vous paraît peut-être évident mais le protocole **UDP** [1], par exemple, envoie des informations au travers du réseau sans se soucier de savoir si elles seront bien réceptionnées par la cible. Ce protocole n'est pas connecté, une application envoie quelque chose au travers du réseau en spécifiant une cible. Il suffit alors de prier très fort pour que le message soit réceptionné correctement !

Plus sérieusement, ce type de protocole est utile si vous avez besoin de faire transiter beaucoup d'informations au travers du réseau mais qu'une petite perte occasionnelle d'informations n'est pas très handicapante. On trouve ce type de protocole dans des jeux graphiques en réseau, le serveur envoyant très fréquemment des informations au client pour qu'il actualise sa fenêtre. Cela fait beaucoup à transmettre mais ce n'est pas dramatique s'il y a une petite perte d'informations de temps à autre puisque, quelques millisecondes plus tard, le serveur renverra de nouveau les informations.

En attendant, c'est le protocole **TCP** qui nous intéresse. Il est un peu plus lent que le protocole **UDP** mais plus sûr et, pour la quantité d'informations que nous allons transmettre, il est préférable d'être sûr des informations transmises plutôt que de la vitesse de transmission.

Clients et serveur

Dans l'architecture que nous allons voir dans ce chapitre, on trouve en général un serveur et plusieurs clients. Le serveur, c'est une machine qui va traiter les requêtes du

1. *User Datagram Protocol*

client.

Si vous accédez par exemple au Site du Zéro, c'est parce que votre navigateur, faisant office de client, se connecte au serveur du Site du Zéro. Il lui envoie un message en lui demandant la page que vous souhaitez afficher et le serveur du Site du Zéro, dans sa grande bonté, envoie la page demandée au client.

Cette architecture est très fréquente, même si ce n'est pas la seule envisageable.

Dans les exemples que nous allons voir, nous allons créer deux applications : l'application **serveur** et l'application **client**. Le serveur *écoute* donc en attendant des connexions et les clients se connectent au serveur.

Les différentes étapes

Nos applications vont fonctionner selon un schéma assez similaire. Voici dans l'ordre les étapes du client et du serveur. Les étapes sont très simplifiées, la plupart des serveurs peuvent communiquer avec plusieurs clients mais nous ne verrons pas cela tout de suite.

Le serveur :

1. attend une connexion de la part du client ;
2. accepte la connexion quand le client se connecte ;
3. échange des informations avec le client ;
4. ferme la connexion.

Le client :

1. se connecte au serveur ;
2. échange des informations avec le serveur ;
3. ferme la connexion.

Comme on l'a vu, le serveur peut dialoguer avec plusieurs clients : c'est tout l'intérêt. Si le serveur du Site du Zéro ne pouvait dialoguer qu'avec un seul client à la fois, il faudrait attendre votre tour, peut-être assez longtemps, avant d'avoir accès à vos pages. Et, sans serveur pouvant dialoguer avec plusieurs clients, les jeux en réseau ou les logiciels de messagerie instantanée seraient bien plus complexes.

Établir une connexion

Pour que le client se connecte au serveur, il nous faut deux informations :

- Le **nom d'hôte** (*host name* en anglais), qui identifie une machine sur Internet ou sur un réseau local. Les noms d'hôtes permettent de représenter des adresses IP de façon plus claire (on a un nom comme `google.fr`, plus facile à retenir que l'adresse IP correspondante `74.125.224.84`).

– Un numéro de port, qui est souvent propre au type d'information que l'on va échanger. Si on demande une connexion web, le navigateur va en général interroger le port 80 si c'est en `http` ou le port 443 si c'est en connexion sécurisée (`https`). Le numéro de port est compris entre 0 et 65535 (il y en a donc un certain nombre !) et les numéros entre 0 et 1023 sont réservés par le système. On peut les utiliser, mais ce n'est pas une très bonne idée.

Pour résumer, quand votre navigateur tente d'accéder au Site du Zéro, il établit une connexion avec le serveur dont le nom d'hôte est `siteduzero.com` sur le port 80. Dans ce chapitre, nous allons plus volontiers travailler avec des noms d'hôtes qu'avec des adresses IP.

Les sockets

Comme on va le voir, les **sockets** sont des objets qui permettent d'ouvrir une connexion avec une machine locale ou distante et d'échanger avec elle.

Ces objets sont définis dans le module `socket` et nous allons maintenant voir comment ils fonctionnent.

Les sockets

Commençons donc, dans la joie et la bonne humeur, par importer notre module `socket`.

```
import socket
```

Nous allons d'abord créer notre serveur puis, en parallèle, un client. Nous allons faire communiquer les deux. Pour l'instant, nous nous occupons du serveur.

Construire notre socket

Nous allons pour cela faire appel au constructeur `socket`. Dans le cas d'une connexion **TCP**, il prend les deux paramètres suivants, dans l'ordre :

– `socket.AF_INET` : la famille d'adresses, ici ce sont des adresses Internet ;
– `socket.SOCK_STREAM` : le type du socket, `SOCK_STREAM` pour le protocole TCP.

```
>>> connexion_principale = socket.socket(socket.AF_INET, socket
    .SOCK_STREAM)
>>>
```

Connecter le socket

Ensuite, nous connectons notre socket. Pour une connexion serveur, qui va attendre des connexions de clients, on utilise la méthode `bind`. Elle prend un paramètre : le tuple `(nom_hote, port)`.



Attends un peu, je croyais que c'était notre client qui se connectait à notre serveur, pas l'inverse...

Oui mais, pour que notre serveur écoute sur un port, il faut le configurer en conséquence. Donc, dans notre cas, le nom de l'hôte sera vide et le port sera celui que vous voulez, entre 1024 et 65535.

```
>>> connexion_principale.bind(('', 12800))
>>>
```

Faire écouter notre socket

Bien. Notre socket est prêt à écouter sur le port 12800 mais il n'écoute pas encore. On va avant tout lui préciser le nombre maximum de connexions qu'il peut recevoir sur ce port sans les accepter. On utilise pour cela la méthode `listen`. On lui passe généralement 5 en paramètre.



Cela veut dire que notre serveur ne pourra dialoguer qu'avec 5 clients maximum ?

Non. Cela veut dire que si 5 clients se connectent et que le serveur n'accepte aucune de ces connexions, aucun autre client ne pourra se connecter. Mais généralement, très peu de temps après que le client ait demandé la connexion, le serveur l'accepte. Vous pouvez donc avoir bien plus de clients connectés, ne vous en faites pas.

```
>>> connexion_principale.listen(5)
>>>
```

Accepter une connexion venant du client

Enfin, dernière étape, on va accepter une connexion. Aucune connexion ne s'est encore présentée mais la méthode `accept` que nous allons utiliser va bloquer le programme tant qu'aucun client ne s'est connecté.

Il est important de noter que la méthode `accept` renvoie deux informations :

- le socket connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client tout juste connecté ;
- un tuple représentant l'adresse IP et le port de connexion du client.



Le port de connexion du client... ce n'est pas le même que celui du serveur ?

Non car votre client, en ouvrant une connexion, passe par un port dit « de sortie » qui va être choisi par le système parmi les ports disponibles. Pour schématiser, quand un client se connecte à un serveur, il emprunte un port (une forme de porte, si vous voulez) puis établit la connexion sur le port du serveur. Il y a donc deux ports dans notre histoire mais celui qu'utilise le client pour ouvrir sa connexion ne va pas nous intéresser.

```
>>> connexion_avec_client, infos_connexion =
    connexion_principale.accept()
```

Cette méthode, comme vous le voyez, bloque le programme. Elle attend qu'un client se connecte. Laissons cette fenêtre Python ouverte et, à présent, ouvrons-en une nouvelle pour construire notre client.

Création du client

Commencez par construire votre socket de la même façon :

```
>>> import socket
>>> connexion_avec_serveur = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
>>>
```

Connecter le client

Pour se connecter à un serveur, on va utiliser la méthode `connect`. Elle prend en paramètre un tuple, comme `bind`, contenant le nom d'hôte et le numéro du port identifiant le serveur auquel on veut se connecter.

Le numéro du port sur lequel on veut se connecter, vous le connaissez : c'est `12800`. Vu que nos deux applications Python sont sur la même machine, le nom d'hôte va être `localhost`[2].

```
>>> connexion_avec_serveur.connect(('localhost', 12800))
>>>
```

Et voilà, notre serveur et notre client sont connectés !

Si vous retournez dans la console Python abritant le serveur, vous pouvez constater que la méthode `accept` ne bloque plus, puisqu'elle vient d'accepter la connexion demandée par le client. Vous pouvez donc de nouveau saisir du code côté serveur :

```
>>> print(infos_connexion)
('127.0.0.1', 2901)
>>>
```

2. C'est-à-dire la machine locale.

La première information, c'est l'adresse IP du client. Ici, elle vaut `127.0.0.1` c'est-à-dire l'IP de l'ordinateur local. Dites-vous que l'hôte `localhost` redirige vers l'IP `127.0.0.1`.

Le second est le port de sortie du client, qui ne nous intéresse pas ici.

Faire communiquer nos sockets

Bon, maintenant, comment faire communiquer nos sockets ? Eh bien, en utilisant les méthodes `send` pour envoyer et `recv` pour recevoir.



Les informations que vous transmettrez seront des chaînes de bytes, pas des `str` !

Donc côté serveur :

```
>>> connexion_avec_client.send(b"Je viens d'accepter la
    connexion")
32
>>>
```

La méthode `send` vous renvoie le nombre de caractères envoyés.

Maintenant, côté client, on va réceptionner le message que l'on vient d'envoyer. La méthode `recv` prend en paramètre le nombre de caractères à lire. Généralement, on lui passe la valeur `1024`. Si le message est plus grand que 1024 caractères, on récupérera le reste après.

Dans la fenêtre Python côté client, donc :

```
>>> msg_recu = connexion_avec_serveur.recv(1024)
>>> msg_recu
b"Je viens d'accepter la connexion"
>>>
```

Magique, non ? Vraiment pas ? Songez que ce petit mécanisme peut servir à faire communiquer des applications entre elles non seulement sur la machine locale, mais aussi sur des machines distantes et reliées par Internet.

Le client peut également envoyer des informations au serveur et le serveur peut les réceptionner, tout cela grâce aux méthodes `send` et `recv` que nous venons de voir.

Fermer la connexion

Pour fermer la connexion, il faut appeler la méthode `close` de notre socket.

Côté serveur :

```
>>> connexion_avec_client.close()
>>>
```

Et côté client :

```
>>> connexion_avec_serveur.close()
>>>
```

Voilà ! Je vais récapituler en vous présentant dans l'ordre un petit serveur et un client que nous pouvons utiliser. Et pour finir, je vous montrerai une façon d'optimiser un peu notre serveur en lui permettant de gérer plusieurs clients à la fois.

Le serveur

Pour éviter les confusions, je vous remets ici le code du serveur, légèrement amélioré. Il n'accepte qu'un seul client (nous verrons plus bas comment en accepter plusieurs) et il tourne jusqu'à recevoir du client le message `fin`.

À chaque fois que le serveur reçoit un message, il envoie en retour le message `'5 / 5'`.

```
import socket

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port)
    )

connexion_avec_client, infos_connexion = connexion_principale.
    accept()

msg_recu = b""
while msg_recu != b"fin":
    msg_recu = connexion_avec_client.recv(1024)
    # L'instruction ci-dessous peut lever une exception si le
        message
    # Réceptionné comporte des accents
    print(msg_recu.decode())
    connexion_avec_client.send(b"5 / 5")

print("Fermeture de la connexion")
connexion_avec_client.close()
connexion_principale.close()
```

Voilà pour le serveur. Il est minimal, vous en conviendrez, mais il est fonctionnel. Nous verrons un peu plus loin comment l'améliorer.

Le client

Là encore, je vous propose le code du client pouvant interagir avec notre serveur.

Il va tenter de se connecter sur le port `12800` de la machine locale. Il demande à l'utilisateur de saisir quelque chose au clavier et envoie ce quelque chose au serveur, puis attend sa réponse.

```
import socket

hote = "localhost"
port = 12800

connexion_avec_serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connexion_avec_serveur.connect((hote, port))
print("Connexion établie avec le serveur sur le port {}".format(port))

msg_a_envoyer = b""
while msg_a_envoyer != b"fin":
    msg_a_envoyer = input("> ")
    # Peut planter si vous tapez des caractères spéciaux
    msg_a_envoyer = msg_a_envoyer.encode()
    # On envoie le message
    connexion_avec_serveur.send(msg_a_envoyer)
    msg_recu = connexion_avec_serveur.recv(1024)
    print(msg_recu.decode()) # Là encore, peut planter s'il y a des accents

print("Fermeture de la connexion")
connexion_avec_serveur.close()
```



Que font les méthodes encode et decode ?

`encode` est une méthode de `str`. Elle peut prendre en paramètre un nom d'encodage et permet de passer un `str` en chaîne **bytes**. C'est, comme vous le savez, ce type de chaîne que `send` accepte. En fait, `encode` *encode* la chaîne `str` en fonction d'un encodage précis (par défaut, **Utf-8**).

`decode`, à l'inverse, est une méthode de **bytes**. Elle aussi peut prendre en paramètre un encodage et elle renvoie une chaîne `str` décodée grâce à l'encodage (par défaut **Utf-8**).

Si l'encodage de votre console est différent d'**Utf-8** (ce sera souvent le cas sur Windows),

des erreurs peuvent se produire si les messages que vous encodez ou décodez comportent des accents.

Voilà, nous avons vu un serveur et un client, tous deux très simples. Maintenant, voyons quelque chose de plus élaboré !

Un serveur plus élaboré



Quel sont les problèmes de notre serveur ?

Si vous y réfléchissez, il y en a pas mal !

- D'abord, notre serveur ne peut accepter qu'un seul client. Si d'autres clients veulent se connecter, ils ne peuvent pas.
- Ensuite, on part toujours du principe qu'on attend le message d'un client et qu'on lui renvoie immédiatement après réception. Mais ce n'est pas toujours le cas : parfois vous envoyez un message au client alors qu'il ne vous a rien envoyé, parfois vous recevez des informations de sa part alors que vous ne lui avez rien envoyé.
  Prenez un logiciel de messagerie instantanée : est-ce que, pour dialoguer, vous êtes obligés d'attendre que votre interlocuteur vous réponde ? Ce n'est pas « j'envoie un message, il me répond, je lui réponds, il me répond »... Parfois, souvent même, vous enverrez deux messages à la suite, peut-être même trois, ou l'inverse, qui sait ?
  Bref, on doit pouvoir envoyer plusieurs messages au client et réceptionner plusieurs messages dans un ordre inconnu. Avec notre technique, c'est impossible (faites le test si vous voulez).

En outre, les erreurs sont assez mal gérées, vous en conviendrez.

Le module `select`

Le module `select` va nous permettre une chose très intéressante, à savoir interroger plusieurs clients dans l'attente d'un message à réceptionner, sans paralyser notre programme.

Pour schématiser, `select` va écouter sur une liste de clients et retourner au bout d'un temps précisé. Ce que renvoie `select`, c'est la liste des clients qui ont un message à réceptionner. Il suffit de parcourir ces clients, de lire les messages en attente (grâce à `recv`) et le tour est joué.

Sur Linux, `select` peut être utilisé sur autre chose que des sockets mais, cette fonctionnalité n'étant pas portable, je ne fais que la mentionner ici.

En théorie

La fonction qui nous intéresse porte le même nom que le module associé, `select`. Elle prend trois ou quatre arguments et en renvoie trois. C'est maintenant qu'il faut être attentif :

Les arguments que prend la fonction sont :

- `rlist` : la liste des sockets en attente d'être lus ;
- `wlist` : la liste des sockets en attente d'être écrits ;
- `xlist` : la liste des sockets en attente d'une erreur (je ne m'attarderai pas sur cette liste) ;
- `timeout` : le délai pendant lequel la fonction attend avant de retourner. Si vous précisez en timeout `0`, la fonction retourne immédiatement. Si ce paramètre n'est pas précisé, la fonction retourne dès qu'un des sockets change d'état (est prêt à être lu s'il est dans `rlist` par exemple) mais pas avant.

Concrètement, nous allons surtout nous intéresser au premier et au quatrième paramètre. En effet, `wlist` et `xlist` ne nous intéresseront pas présentement.

Ce qu'on veut, c'est mettre des sockets dans une liste et que `select` les surveille, en retournant dès qu'un socket est prêt à être lu. Comme cela notre programme ne bloque pas et il peut recevoir des messages de plusieurs clients dans un ordre complètement inconnu.

Maintenant, concernant le `timeout` : comme je vous l'ai dit, si vous ne le précisez pas, `select` bloque jusqu'au moment où l'un des sockets que nous écoutons est prêt à être lu, dans notre cas. Si vous précisez un `timeout` de `0`, `select` retournera tout de suite. Sinon, `select` retournera au bout du temps que vous indiquez en secondes, ou plus tôt si un socket est prêt à être lu.

En gros, si vous précisez un `timeout` de `1`, la fonction va bloquer pendant une seconde maximum. Mais si un des sockets en écoute est prêt à être lu dans l'intervalle (c'est-à-dire si un des clients envoie un message au serveur), la fonction retourne prématurément.

`select` renvoie trois listes, là encore `rlist`, `wlist` et `xlist`, sauf qu'il ne s'agit pas des listes fournies en entrée mais uniquement des sockets « à lire » dans le cas de `rlist`.

Ce n'est pas clair ? Considérez cette ligne (ne l'essayez pas encore) :

```
rlist, wlist, xlist = select.select(clients_connectes, [], [],
    2)
```

Cette instruction va écouter les sockets contenus dans la liste `clients_connectes`. Elle retournera au plus tard dans 2 secondes. Mais elle retournera plus tôt si un client envoie un message. La liste des clients ayant envoyé un message se retrouve dans notre variable `rlist`. On la parcourt ensuite et on peut appeler `recv` sur chacun des sockets.

Si ce n'est pas plus clair, rassurez-vous : nous allons voir `select` en action un peu plus bas. Vous pouvez également aller jeter un coup d'œil à la documentation du module, avec le code web suivant :

▷ Module select
Code web : 261326

select en action

Nous allons un peu travailler sur notre serveur. Vous pouvez garder le même client de test.

Le but va être de créer un serveur pouvant accepter plusieurs clients, réceptionner leurs messages et leur envoyer une confirmation à chaque réception. L'exercice ne change pas beaucoup mais on va utiliser `select` pour travailler avec plusieurs clients.

J'ai parlé de `select` pour écouter plusieurs clients connectés mais cette fonction va également nous permettre de savoir si un (ou plusieurs) clients sont connectés au serveur. Si vous vous souvenez, la méthode `accept` est aussi une fonction bloquante. On va du reste l'utiliser de la même façon qu'un peu plus haut.

Je crois vous avoir donné assez d'informations théoriques. Le code doit parler maintenant :

```
import socket
import select

hote = ''
port = 12800

connexion_principale = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
print("Le serveur écoute à présent sur le port {}".format(port)
    )

serveur_lance = True
clients_connectes = []
while serveur_lance:
    # On va vérifier que de nouveaux clients ne demandent pas à
        se connecter
    # Pour cela, on écoute la connexion_principale en lecture
    # On attend maximum 50ms
    connexions_demandees, wlist, xlist = select.select([
        connexion_principale],
        [], [], 0.05)

    for connexion in connexions_demandees:
        connexion_avec_client, infos_connexion = connexion.
            accept()
        # On ajoute le socket connecté à la liste des clients
        clients_connectes.append(connexion_avec_client)

    # Maintenant, on écoute la liste des clients connectés
```

```
    # Les clients renvoyés par select sont ceux devant être lus (recv)
    # On attend là encore 50ms maximum
    # On enferme l'appel à select.select dans un bloc try
    # En effet, si la liste de clients connectés est vide, une exception
    # Peut être levée
    clients_a_lire = []
    try:
        clients_a_lire, wlist, xlist = select.select(clients_connectes,
                [], [], 0.05)
    except select.error:
        pass
    else:
        # On parcourt la liste des clients à lire
        for client in clients_a_lire:
            # Client est de type socket
            msg_recu = client.recv(1024)
            # Peut planter si le message contient des caractères spéciaux
            msg_recu = msg_recu.decode()
            print("Reçu {}".format(msg_recu))
            client.send(b"5 / 5")
            if msg_recu == "fin":
                serveur_lance = False

print("Fermeture des connexions")
for client in clients_connectes:
    client.close()

connexion_principale.close()
```

C'est plus long hein ? C'est inévitable, cependant.

Maintenant notre serveur peut accepter des connexions de plus d'un client, vous pouvez faire le test. En outre, il ne se bloque pas dans l'attente d'un message, du moins pas plus de 50 millisecondes.

Je pense que les commentaires sont assez précis pour vous permettre d'aller plus loin. Ceci n'est naturellement pas encore une version complète mais, grâce à cette base, vous devriez pouvoir facilement arriver à quelque chose. Pourquoi ne pas faire un mini tchat ?

Les déconnexions fortuites ne sont pas gérées non plus. Mais vous avez assez d'éléments pour faire des tests et améliorer notre serveur si cela vous tente.

Et encore plus

Je vous l'ai dit, le réseau est un vaste sujet et, même en se restreignant au sujet que j'ai choisi, il y aurait beaucoup d'autres choses à vous montrer. Je ne peux tout simplement pas remplacer la documentation et donc, si vous voulez en apprendre plus, je vous invite à jeter un coup d'œil aux codes web suivants :

▷ Module socket
Code web : 695428

▷ Module select
Code web : 261326

▷ Module socketserver
Code web : 672065

Le dernier module, `socketserver`, propose une alternative pour monter vos applications serveur. Il en existe d'autres, dans tous les cas : vous pouvez utiliser des sockets non bloquants (c'est-à-dire qui ne bloquent pas le programme quand vous utilisez leur méthode `accept` ou `recv`) ou des **threads** pour exécuter différentes portions de votre programme en parallèle. Mais je vous laisse vous documenter sur ces sujets s'ils vous intéressent !

En résumé

– Dans la structure réseau que nous avons vue, on trouve un **serveur** pouvant dialoguer avec plusieurs **clients**.
– Pour créer une connexion côté serveur ou client, on utilise le module `socket` et la classe `socket` de ce module.
– Pour se connecter à un serveur, le socket client utilise la méthode `connect`.
– Pour écouter sur un port précis, le serveur utilise d'abord la méthode `bind` puis la méthode `listen`.
– Pour s'échanger des informations, les sockets client et serveur utilisent les méthodes `send` et `recv`.
– Pour fermer une connexion, le socket serveur ou client utilise la méthode `close`.
– Le module `select` peut être utile si l'on souhaite créer un serveur pouvant gérer plusieurs connexions simultanément ; toutefois, il en existe d'autres.

Chapitre 31

Des interfaces graphiques avec Tkinter

Difficulté : 

Nous allons maintenant voir comment créer des interfaces graphiques à l'aide d'un module présent par défaut dans Python : **Tkinter**.

Ce module permet de créer des interfaces graphiques en offrant une passerelle entre Python et la bibliothèque **Tk**.

Vous allez pouvoir apprendre dans ce chapitre à créer des fenêtres, créer des boutons, faire réagir vos objets graphiques à certains évènements...



Présentation de Tkinter

Tkinter (**Tk interface**) est un module intégré à la bibliothèque standard de Python, bien qu'il ne soit pas maintenu directement par les développeurs de Python. Il offre un moyen de créer des interfaces graphiques *via* Python.

Tkinter est disponible sur Windows et la plupart des systèmes Unix. Les interfaces que vous pourrez développer auront donc toutes les chances d'être portables d'un système à l'autre.

Notez qu'il existe d'autres bibliothèques pour créer des interfaces graphiques. **Tkinter** a l'avantage d'être disponible par défaut, sans nécessiter une installation supplémentaire.

Pour savoir si vous pouvez utiliser le module **Tkinter** *via* la version de Python installée sur votre système, tapez dans l'interpréteur en ligne de commande Python :

```
from tkinter import *
```

Si une erreur se produit, vous devrez aller vous renseigner sur la page du Wiki Python consacrée à **Tkinter**, grâce au code web suivant :

▷ Wiki Tkinter
Code web : 218447

Votre première interface graphique

Nous allons commencer par voir le code minimal pour créer une fenêtre avec **Tkinter**. Petit à petit, nous allons apprendre à rajouter des choses, mais commençons par voir la base de code que l'on retrouve d'une interface **Tkinter** à l'autre.

Étant en Python, ce code minimal est plutôt court :

```
"""Premier exemple avec Tkinter.

On crée une fenêtre simple qui souhaite la bienvenue à l'
    utilisateur.

"""

# On importe Tkinter
from tkinter import *

# On crée une fenêtre, racine de notre interface
fenetre = Tk()

# On crée un label (ligne de texte) souhaitant la bienvenue
# Note : le premier paramètre passé au constructeur de Label
    est notre
# interface racine
champ_label = Label(fenetre, text="Salut les Zér0s !")

```

```
# On affiche le label dans la fenêtre
champ_label.pack()

# On démarre la boucle Tkinter qui s'interompt quand on ferme
    la fenêtre
fenetre.mainloop()
```

Vous pouvez voir le résultat à la figure 31.1.

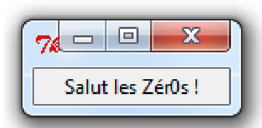


FIGURE 31.1 – Notre première fenêtre avec Tkinter

Vous pouvez recopier ce code dans un fichier `.py` (n'oubliez pas de rajouter la ligne spécifiant l'encodage). Vous pouvez ensuite exécuter votre programme, ce qui affiche une fenêtre (simple, certes, mais une fenêtre tout de même).

Comme vous pouvez le voir, la fenêtre est tout juste assez grande pour que le message s'affiche.

Regardons le code d'un peu plus près.

1. On commence par importer **Tkinter**, sans grande surprise.
2. On crée ensuite un objet de la classe `Tk`. La plupart du temps, cet objet sera la fenêtre principale de notre interface.
3. On crée un `Label`, c'est-à-dire un objet graphique affichant du texte
4. On appelle la méthode `pack` de notre `Label`. Cette méthode permet de positionner l'objet dans notre fenêtre (et, par conséquent, de l'afficher).
5. Enfin, on appelle la méthode `mainloop` de notre fenêtre racine. Cette méthode ne retourne que lorsqu'on ferme la fenêtre.

Quelques petites précisions :

– Nos objets graphiques (boutons, champs de texte, cases à cocher, barres de progression...) sont appelés des **widgets**.
– On peut préciser plusieurs options lors de la construction de nos **widgets**. Ici, on définit l'option `text` de notre `Label` à `"Salut les Zér0s !"`.

Il existe d'autres options communes à la plupart des widgets (la couleur de fond `bg`, la couleur du widget `fg`, etc.) et d'autres plus spécifiques à un certain type de widget. Le `Label` par exemple possède l'option `text` représentant le texte affiché par le `Label`.

Comme nous l'avons vu, vous pouvez modifier des options lors de la création du widget. Mais vous pouvez aussi en modifier après :

```
>>> champ_label["text"]
'Salut les Zér0s !'
>>> champ_label["text"] = "Maintenant, au revoir !"
>>> champ_label["text"]
'Maintenant, au revoir !'
>>>
```

Comme vous le voyez, vous passez entre crochets (comme pour accéder à une valeur d'un dictionnaire) le nom de l'option. C'est le même principe pour accéder à la valeur actuelle de l'option ou pour la modifier.

Nous allons voir quelques autres widgets de **Tkinter** à présent.

De nombreux widgets

Tkinter définit un grand nombre de widgets pouvant être utilisés dans notre fenêtre. Nous allons en voir ici quelques-uns.

Les widgets les plus communs

Les labels

C'est le premier widget que nous avons vu, hormis notre fenêtre principale qui en est un également. On s'en sert pour afficher du texte dans notre fenêtre, du texte qui ne sera pas modifié par l'utilisateur.

```
champ_label = Label(fenetre, text="contenu de notre champ label
    ")
champ_label.pack()
```

N'oubliez pas que, pour qu'un widget apparaisse, il faut :

– qu'il prenne, en premier paramètre du constructeur, la fenêtre principale ;
– qu'il fasse appel à la méthode `pack`.

La méthode `pack` permet de positionner un objet dans une fenêtre ou dans un cadre, nous verrons plus loin quelques-uns de ses paramètres optionnels.

Les boutons

Les boutons sont des widgets sur lesquels on peut cliquer et qui peuvent déclencher des actions ou **commandes** comme nous le verrons ultérieurement plus en détail.

```
bouton_quitter = Button(fenetre, text="Quitter", command=
    fenetre.quit)
bouton_quitter.pack()
```

J'imagine que vous vous posez des questions sur le dernier paramètre passé à notre constructeur de `Button`. Il s'agit de l'action liée à un clic sur le bouton. Ici, c'est la méthode `quit` de notre fenêtre racine qui est appelée.

Ainsi, quand vous cliquez sur le bouton `Quitter`, la fenêtre se ferme. Nous verrons plus tard comment créer nos propres commandes.



Si vous faites des tests depuis l'interpréteur Python en ligne de commande, la fenêtre **Tk** reste ouverte tant que la console reste ouverte. Le bouton `Quitter` interrompra la boucle `mainloop` mais ne fermera pas l'interface.

Une ligne de saisie

Le widget que nous allons voir à présent est une zone de texte dans lequel l'utilisateur peut écrire. En fait de zone, il s'agit d'une ligne simple.

On préférera créer une variable **Tkinter** associée au champ de texte. Regardez le code qui suit :

```
1 var_texte = StringVar()
2 ligne_texte = Entry(fenetre, textvariable=var_texte, width=30)
3 ligne_texte.pack()
```

À la ligne 1, nous créons une variable **Tkinter**. En résumé, c'est une variable qui va ici contenir le texte de notre `Entry`. Il est possible de lier cette variable à une méthode de telle sorte que la méthode soit appelée quand la variable est modifiée (l'utilisateur écrit dans le champ `Entry`.

Pour en savoir plus, je vous renvoie à la méthode `trace` de la variable.

Comme vous l'avez peut-être remarqué, le widget `Entry` n'est qu'une zone de saisie. Pour que l'utilisateur sache ce qu'il doit y écrire, il pourrait être utile de lui mettre une indication auprès du champ. Le widget `Label` est le plus approprié dans ce cas.

Notez qu'il existe également le widget `Text` qui représente un champ de texte à plusieurs lignes.

Les cases à cocher

Les cases à cocher sont définies dans la classe `Checkbutton`. Là encore, on utilise une variable pour surveiller la sélection de la case.

Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive), on préférera créer une variable de type `IntVar` plutôt que `StringVar`, bien que ce ne soit pas une obligation.

```
var_case = IntVar()
case = Checkbutton(fenetre, text="Ne plus poser cette question"
    , variable=var_case)
case.pack()
```

Vous pouvez ensuite contrôler l'état de la case à cocher en interrogeant la variable :

```
var_case.get()
```

Si la case est cochée, la valeur renvoyée par la variable sera `1`. Si elle n'est pas cochée, ce sera `0`.

Notez qu'à l'instar d'un bouton, vous pouvez lier la case à cocher à une commande qui sera appelée quand son état change.

Les boutons radio

Les boutons radio (*radio buttons* en anglais) sont des boutons généralement présentés en groupes. C'est, à proprement parler, un ensemble de cases à cocher mutuellement exclusives : quand vous cliquez sur l'un des boutons, celui-ci se sélectionne et tous les autres boutons du même groupe se désélectionnent.

Ce type de bouton est donc surtout utile dans le cadre d'un regroupement.

Pour créer un groupe de boutons, il faut simplement qu'ils soient tous associés à la même variable (là encore, une variable **Tkinter**). La variable peut posséder le type que vous voulez.

Quand l'utilisateur change le bouton sélectionné, la valeur de la variable change également en fonction de l'option `value` associée au bouton. Voyons un exemple :

```
var_choix = StringVar()

choix_rouge = Radiobutton(fenetre, text="Rouge", variable=
    var_choix, value="rouge")
choix_vert = Radiobutton(fenetre, text="Vert", variable=
    var_choix, value="vert")
choix_bleu = Radiobutton(fenetre, text="Bleu", variable=
    var_choix, value="bleu")

choix_rouge.pack()
choix_vert.pack()
choix_bleu.pack()
```

Pour récupérer la valeur associée au bouton actuellement sélectionné, interrogez la variable :

```
var_choix.get()
```

Les listes déroulantes

Ce widget permet de construire une liste dans laquelle on peut sélectionner un ou plusieurs éléments. Le fonctionnement n'est pas tout à fait identique aux boutons radio. Ici, la liste comprend plusieurs lignes et non un groupe de boutons.

Créer une liste se fait assez simplement, vous devez commencer à vous habituer à la syntaxe :

```
liste = Listbox(fenetre)
liste.pack()
```

On insère ensuite des éléments. La méthode `insert` prend deux paramètres :

1. la position à laquelle insérer l'élément ;
2. l'élément même, sous la forme d'une chaîne de caractères.

Si vous voulez insérer des éléments à la fin de la liste, utilisez la constante `END` définie par **Tkinter** :

```
liste.insert(END, "Pierre")
liste.insert(END, "Feuille")
liste.insert(END, "Ciseau")
```

Pour accéder à la sélection, utilisez la méthode `curselection` de la liste. Elle renvoie un **tuple** de chaînes de caractères, chacune étant la position de l'élément sélectionné.

Par exemple, si `liste.curselection()` renvoie `('2',)`, c'est le troisième élément de la liste qui est sélectionné (`Ciseau` en l'occurrence).

Organiser ses widgets dans la fenêtre

Il existe plusieurs widgets qui peuvent contenir d'autres widgets. L'un d'entre eux se nomme `Frame`. C'est un cadre rectangulaire dans lequel vous pouvez placer vos widgets... ainsi que d'autres objets `Frame` si besoin est.

Si vous voulez qu'un widget apparaisse dans un cadre, utilisez le `Frame` comme parent à la création du widget :

```
cadre = Frame(fenetre, width=768, height=576, borderwidth=1)
cadre.pack(fill=BOTH)

message = Label(cadre, text="Notre fenêtre")
message.pack(side="top", fill=X)
```

Comme vous le voyez, nous avons passé plusieurs arguments nommés à notre méthode `pack`. Cette méthode, je vous l'ai dit, sert à placer nos widgets dans la fenêtre (ici, dans le cadre).

En précisant `side="top"`, on demande à ce que le widget soit placé en haut de son parent (ici, notre cadre).

Il existe aussi l'argument nommé `fill` qui permet au widget de remplir le widget parent, soit en largeur si la valeur est `X`, soit en hauteur si la valeur est `Y`, soit en largeur et hauteur si la valeur est `BOTH`.

D'autres arguments nommés existent, bien entendu. Si vous voulez une liste exhaustive, rendez-vous sur le chapitre consacré à **Tkinter** dans la documentation officielle de Python, accessible avec le code web suivant :

▷ Documentation Tkinter
Code web : 891716

Une partie est consacrée au **packer** et à la méthode `pack`.

Notez qu'il existe aussi le widget `Labelframe`, un cadre avec un titre, ce qui nous évite d'avoir à placer un `label` en haut du cadre. Il se construit comme un `Frame` mais peut prendre en argument, à la construction, le texte représentant le titre : `cadre = Labelframe(..., text="Titre du cadre")`

Bien d'autres widgets

Vous devez vous en douter, ceci n'est qu'une approche très sommaire de quelques widgets de **Tkinter**. Il en existe de nombreux autres et ceux que nous avons vus ont bien d'autres options.

Il est notamment possible de créer une barre de menus avec ses menus imbriqués, d'afficher des images, des **canvas** dans lequel vous pouvez dessiner pour personnaliser votre fenêtre... bref, il vous reste bien des choses à voir, même si ce chapitre ne peut pas couvrir tous ces widgets et options.

Je vous propose pour l'heure d'aller jeter un coup d'œil sur les commandes que nous avons effleurées jusqu'ici sans trop nous pencher dessus.

Les commandes

Nous avons vu très brièvement comment faire en sorte qu'un bouton ferme une fenêtre quand on clique dessus :

```
bouton_quitter = Button(fenetre, text="Quitter", command=
    fenetre.quit)
```

C'est le dernier argument qui est important ici. Il a pour nom `command` et a pour valeur la méthode `quit` de notre fenêtre.

Sur ce modèle, nous pouvons créer assez simplement des commandes personnalisées, en écrivant des méthodes.

Cependant, il y a ici une petite subtilité : la méthode que nous devons créer ne prend aucun paramètre. Si nous voulons qu'un clic sur le bouton modifie le bouton lui-même ou un autre objet, nous devons placer nos widgets dans un corps de classe.

D'ailleurs, à partir du moment où on sort du cadre d'un test, il est préférable de mettre le code dans une classe.

On peut la faire hériter de `Frame`, ce qui signifie que notre classe sera un widget elle aussi. Voyons un code complet que j'expliquerai plus bas :

```
from tkinter import *

class Interface(Frame):

    """Notre fenêtre principale.
```

```
    Tous les widgets sont stockés comme attributs de cette fenêtre."""

    def __init__(self, fenetre, **kwargs):
        Frame.__init__(self, fenetre, width=768, height=576, **kwargs)
        self.pack(fill=BOTH)
        self.nb_clic = 0

        # Création de nos widgets
        self.message = Label(self, text="Vous n'avez pas cliqué sur le bouton.")
        self.message.pack()

        self.bouton_quitter = Button(self, text="Quitter", command=self.quit)
        self.bouton_quitter.pack(side="left")

        self.bouton_cliquer = Button(self, text="Cliquez ici", fg="red",
                command=self.cliquer)
        self.bouton_cliquer.pack(side="right")

    def cliquer(self):
        """Il y a eu un clic sur le bouton.

        On change la valeur du label message."""

        self.nb_clic += 1
        self.message["text"] = "Vous avez cliqué {} fois.".format(self.nb_clic)
```

▷ Copier ce code
Code web : 941793

Et pour créer notre interface :

```
fenetre = Tk()
interface = Interface(fenetre)

interface.mainloop()
interface.destroy()
```

La figure 31.2 vous montre le résultat obtenu.

Dans l'ordre :

1. On crée une classe qui contiendra toute la fenêtre. Cette classe hérite de `Frame`, c'est-à-dire d'un cadre **Tkinter**.
2. Dans le constructeur de la fenêtre, on appelle le constructeur du cadre et on `pack` (positionne et affiche) le cadre.

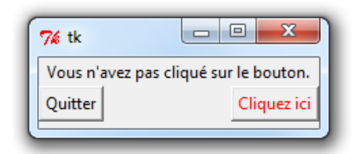


FIGURE 31.2 – La fenêtre est créée avec ses deux boutons

3. Toujours dans le constructeur, on crée les différents widgets de la fenêtre. On les positionne et les affiche également.
4. On crée une méthode `bouton_cliquer`, qui est appelée quand on clique sur le `bouton_cliquer`. Elle ne prend aucun paramètre. Elle va mettre à jour le texte contenu dans le **label** `self.message` pour afficher le nombre de clics enregistrés sur le bouton.
5. On crée la fenêtre `Tk` qui est l'objet parent de l'interface que l'on instancie ensuite.
6. On rentre dans la boucle `mainloop`. Elle s'interrompra quand on fermera la fenêtre.
7. Ensuite, on détruit la fenêtre grâce à la méthode `destroy`.

Pour conclure

Ceci n'est qu'un survol, j'insiste sur ce point. **Tkinter** est une bibliothèque trop riche pour être présentée en un chapitre. Vous trouverez de nombreux exemples d'interfaces de par le Web, si vous cherchez quelque chose de plus précis.

En résumé

- **Tkinter** est un module intégré à la bibliothèque standard et permettant de créer des interfaces graphiques.
- Les objets graphiques (boutons, zones de texte, cases à cocher...) sont appelés des **widgets**.
- Dans **Tkinter**, les **widgets** prennent, lors de leur construction, leur objet parent en premier paramètre.
- Chaque **widget** possède des options qu'il peut préciser comme arguments nommés lors de sa construction.
- On peut également accéder aux options d'un widget ainsi : `widget["nom_option"]`.

Cinquième partie

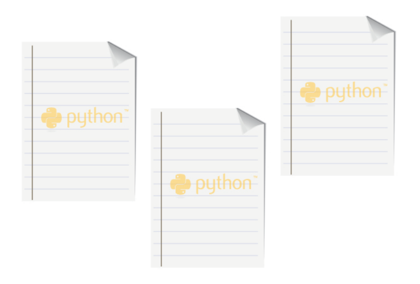
Annexes

Chapitre 32

Écrire nos programmes Python dans des fichiers

Difficulté :

Ce petit chapitre vous explique comment mettre votre code Python dans un fichier pour l'exécuter. Vous pouvez lire ce chapitre très rapidement tant que vous savez à quoi sert la fonction `print`, c'est tout ce dont vous avez besoin.



Mettre le code dans un fichier

Pour placer du code dans un fichier que nous pourrons ensuite exécuter, la démarche est très simple :

1. Ouvrez un éditeur standard sans mise en forme (Notepad++, VIM ou Emacs...). Dans l'absolu, le bloc-notes Windows est aussi candidat mais il reste moins agréable pour programmer (pas de coloration syntaxique du code, notamment).
2. Dans ce fichier, recopiez simplement `print("Bonjour le monde !")`, comme à la figure 32.1.
3. Enregistrez ce code dans un fichier à l'extension `.py`, comme à la figure 32.2. Cela est surtout utile sur Windows.



Je recommande fortement Notepad++ aux utilisateurs de Windows : il est léger, gratuit et très complet.

▷ Télécharger Notepad++
Code web : 830037

FIGURE 32.1 – Une ligne de code dans Notepad++

Exécuter notre code sur Windows

Dans l'absolu, vous pouvez faire un double-clic sur le fichier à l'extension `.py`, dans l'explorateur de fichiers. Mais la fenêtre s'ouvre et se referme très rapidement. Pour éviter cela, vous avez trois possibilités :

- mettre le programme en pause (voir la dernière section de ce chapitre) ;
- lancer le programme depuis la console Windows (je ne m'attarderai pas ici sur cette solution) ;
- exécuter le programme avec **IDLE**.

C'est cette dernière opération que je vais détailler brièvement. Faites un clic droit sur le fichier `.py`. Dans le menu contextuel, vous devriez voir apparaître un intitulé du type `edit with IDLE`. Cliquez dessus.

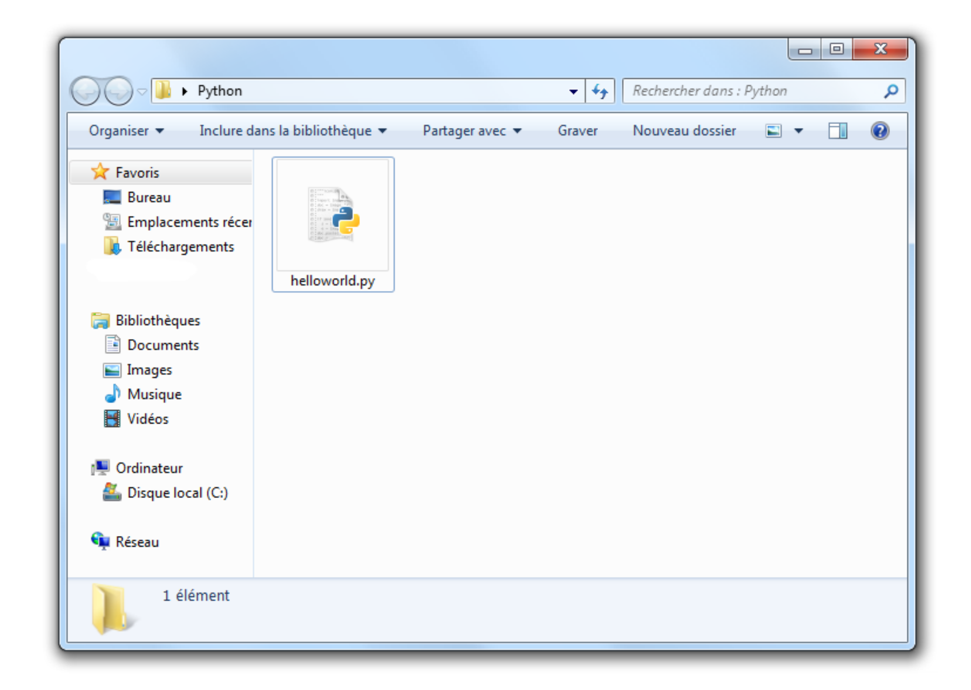


FIGURE 32.2 – Un fichier `.py` sous Windows

La fenêtre d'IDLE devrait alors s'afficher. Vous pouvez voir votre code, ainsi que plusieurs boutons. Cliquez sur `run` puis sur `run module` (ou appuyez sur F5 directement).

Le code du programme devrait alors se lancer. Cette fois, la fenêtre de console reste ouverte pour que vous puissiez voir le résultat ou les erreurs éventuelles.

Sur les systèmes Unix

Il est nécessaire d'ajouter, tout en haut de votre programme, une ligne qui indique le chemin menant vers l'interpréteur. Elle se présente sous la forme : `#!/chemin`.

Les habitués du Bash devraient reconnaître cette ligne assez rapidement. Pour les autres, sachez qu'il suffit de mettre à la place de « `chemin` » le chemin absolu de l'interpréteur (le chemin qui, en partant de la racine du système, mène à l'interpréteur Python). Par exemple :

```
#!/usr/bin/python3.2
```

En changeant les droits d'accès en exécution sur le fichier, vous devriez pouvoir le lancer directement.

Préciser l'encodage de travail

À partir du moment où vous mettez des accents dans votre programme, vous devrez préciser l'encodage que vous utilisez pour l'écrire.

Décrit très brièvement, l'encodage est une table contenant une série de codes symbolisant différents accents. Il existe deux encodages très utilisés : l'encodage **Latin-1** sur Windows et l'encodage **Utf-8** que l'on retrouve surtout sur les machines Unix.

Vous devez préciser à Python dans quel encodage vous écrivez votre programme. La plupart du temps, sur Windows, ce sera donc **Latin-1**, alors que sur Linux et Mac ce sera plus vraisemblablement **Utf-8**.

Une ligne de commentaire doit être ajoutée tout en haut de votre code (si vous êtes sur un système Unix, sous la ligne qui fournit le chemin menant vers l'interpréteur). Cette ligne s'écrit ainsi :

```
# -*-coding:encodage -*
```

Remplacez `encodage` par l'encodage que vous utilisez en fonction de votre système.

Sur Windows, on trouvera donc plus vraisemblablement : `# -*-coding:Latin-1 -*`

Sur Linux ou Mac, ce sera plus vraisemblablement : `# -*-coding:Utf-8 -*`

Gardez la ligne qui fonctionne chez vous et n'oubliez pas de la mettre en tête de chacun de vos fichiers exécutables Python.

Pour en savoir plus sur l'encodage, je vous renvoie au code web suivant, où vous trouverez plein d'informations utiles :

▷ Du Latin-1 à l'Unicode
Code web : 691519

Mettre en pause notre programme

Sur Windows se pose un problème : si vous lancez votre programme en faisant directement un double-clic dessus dans l'explorateur, il aura tendance à s'ouvrir et se fermer très rapidement. Python exécute bel et bien le code et affiche le résultat, mais tout cela très rapidement. Et une fois que la dernière ligne de code a été exécutée, Windows ferme la console.

Pour pallier ce problème, on peut demander à Python de se mettre en pause à la fin de l'exécution du code.

Il va falloir ajouter deux lignes, l'une au début de notre programme et l'autre tout à la fin. La première importe le module `os` et la seconde utilise une fonction de ce module pour mettre en pause le programme. Si vous ne savez pas ce qu'est un module, ne vous en faites pas : le code suffira, un chapitre dans la première partie vous explique ce dont il s'agit.

```
# -*-coding:Latin-1 -*
import os # On importe le module os
```

```
3 print("Bonjour le monde !")
4 os.system("pause")
```

Ce sont les lignes 2 et 4 qui sont nouvelles pour nous et que vous devez retenir. Quand vous exécutez ce code, vous obtenez :

```
Bonjour le monde !
Appuyez sur une touche pour continuer...
```

Vous pouvez donc lancer ce programme en faisant directement un double-clic dessus dans l'explorateur de fichiers.



Notez bien que ce code ne fonctionne que sur Windows !

Si vous voulez mettre en pause votre programme sur Linux ou Mac, vous devrez utiliser un autre moyen. Même si la fonction `input` n'est pas faite pour, vous pouvez l'utiliser conclure votre programme par la ligne :

```
input("Appuyez sur ENTREE pour fermer ce programme...")
```

En résumé

- Pour créer un programme Python, il suffit d'écrire du code dans un fichier (de préférence avec l'extension `.py`).
- Si on utilise des accents dans le code, il est nécessaire de préciser, en tête de fichier, l'encodage utilisé.
- Sur Windows, pour lancer notre programme Python depuis l'explorateur, il faut le mettre en pause grâce au module `os` et à sa fonction `system`.

Chapitre 33

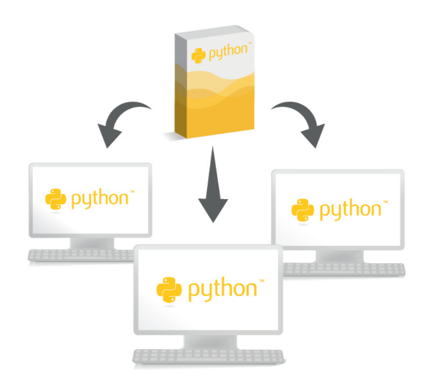
Distribuer facilement nos programmes Python avec cx_Freeze

Difficulté : 

Comme nous l'avons vu, Python nous permet de générer des exécutables d'une façon assez simple. Mais, si vous en venez à vouloir distribuer votre programme, vous risquez de vous heurter au problème suivant : pour lancer votre code, votre destinaire doit installer Python ; qui plus est, la bonne version. Et si vous commencez à utiliser des bibliothèques tierces, il doit aussi les installer !

Heureusement, il existe plusieurs moyens pour produire des fichiers exécutables que vous pouvez distribuer et qui incluent tout le nécessaire.

Sur Windows, il faut enfermer vos fichiers à l'extension `.py` dans un `.exe` accompagné de fichiers `.dll`. **Cx_freeze** est un des outils qui permet d'atteindre cet objectif.



En théorie

L'objectif de ce chapitre est de vous montrer comment faire des programmes dits *standalone*[1]. Comme vous le savez, pour que vos fichiers `.py` s'exécutent, il faut que Python soit installé sur votre machine. Mais vous pourriez vouloir transmettre votre programme sans obliger vos utilisateurs à installer Python sur leur ordinateur.

Une version *standalone* de votre programme contient, en plus de votre code, l'exécutable Python et les dépendances dont il a besoin.

Sur Windows, vous vous retrouverez avec un fichier `.exe` et plusieurs fichiers compagnons, bien plus faciles à distribuer et, pour vos utilisateurs, à exécuter.

Le programme résultant ne sera pas sensiblement plus rapide ou plus lent. Il ne s'agit pas de compilation, Python reste un langage interprété et l'interpréteur sera appelé pour lire votre code, même si celui-ci se trouvera dans une forme un peu plus compressée.

Avantages de cx_Freeze

– Portabilité : **cx_Freeze** est fait pour fonctionner aussi bien sur Windows que sur Linux ou Mac OS ;
– Compatibilité : **cx_Freeze** fonctionne sur des projets Python de la branche 2.X ou 3.X ;
– Simplicité : créer son programme *standalone* avec **cx_Freeze** est simple et rapide ;
– Souplesse : vous pouvez aisément personnaliser votre programme *standalone* avant de le construire.

Il existe d'autres outils similaires, dont le plus célèbre est **py2exe**. Il est accessible via le code web suivant :

▷ Télécharger py2exe
Code web : 603896

Il a toutefois l'inconvénient de ne fonctionner que sur Windows et, à l'heure où j'écris ces lignes du moins, de ne pas proposer de version compatible avec Python 3.X.

Nous allons à présent voir comment installer **cx_Freeze** et comment construire nos programmes *standalone*.

En pratique

Il existe plusieurs façons d'utiliser **cx_Freeze**. Il nous faut dans tous les cas commencer par l'installer.

1. Que l'on peut traduire très littéralement par "se tenir seul".

Installation

Sur Windows

Rendez-vous sur le site `sourceforge`, où est hébergé le projet **cx_Freeze**, en utilisant le code web suivant :

▷ Télécharger cx_Freeze
Code web : 429982

et téléchargez le fichier correspondant à votre version de Python.

Après l'avoir téléchargé, lancez l'exécutable et laissez-vous guider. Rien de trop technique jusqu'ici !

Sur Linux

Je vous conseille d'installer **cx_Freeze** depuis les sources.

Commencez par vous rendre sur le site de téléchargement via le code web ci-dessus et sélectionnez la dernière version de **cx_Freeze** (*Source Code only*).

Téléchargez et décompressez les sources :

```
tar -xvf cx\_Freeze_version.tar.gz
```

Rendez-vous dans le dossier décompressé puis lancez l'installation en tant qu'utilisateur `root` :

```
$ cd cx\_Freeze_version
$ sudo python3.2 setup.py build
$ sudo python3.2 setup.py install
```

Si ces deux commandes s'exécutent convenablement, vous disposerez de la commande `cxfreeze` :

```
$ cxfreeze
cxfreeze: error: script or a list of modules must be specified
```

Utiliser le script `cxfreeze`

Pour les utilisateurs de Windows, je vous invite à vous rendre dans la ligne de commande (`Démarrer` > `Exécuter...` > `cmd`).

Rendez-vous dans le sous-dossier `scripts` de votre installation Python (chez moi, `C:\python32\scripts`).

```
cd \
cd C:\python32\scripts
```

Sur Linux, vous devez avoir accès au script directement. Vous pouvez le vérifier en tapant `cxfreeze` dans la console. Si cette commande n'est pas disponible mais que vous avez installé **cxfreeze**, vous pourrez trouver le script dans le répertoire `bin` de votre version de Python.

Sur Windows ou Linux, la syntaxe du script est la même : `cxfreeze fichier.py`.

Faisons un petit programme pour le tester. Créez un fichier `salut.py` (sur Windows, mettez-le dans le même dossier que le script `cxfreeze`, ce sera plus simple pour le test). Vous pouvez y placer le code suivant :

```
"""Ce fichier affiche simplement une ligne grâce à la fonction
    print."""

import os

print("Salut le monde !")

# Sous Windows il faut mettre ce programme en pause (inutile
    sous Linux)
os.system("pause")
```



N'oubliez pas la ligne spécifiant l'encodage.

À présent, lancez le script **cxfreeze** en lui passant en paramètre le nom de votre fichier : `cxfreeze salut.py`.

Si tout se passe bien, vous vous retrouvez avec un sous-dossier `dist` qui contient les bibliothèques dont votre programme a besoin pour s'exécuter… et votre programme lui-même.

Sur Windows, ce sera `salut.exe`. Sur Linux, ce sera simplement `salut`.

Vous pouvez lancer cet exécutable : comme vous le voyez, votre message s'affiche bien à l'écran.

Formidable ! Ou pas…

Au fond, vous ne voyez sans doute pas de différence avec votre programme `salut.py`. Vous pouvez l'exécuter, lui aussi, il n'y a aucune différence.

Sauf que l'exécutable que vous trouvez dans le sous-dossier `dist` n'a pas besoin de Python pour s'exécuter : il contient lui-même l'interpréteur Python.

Vous pouvez donc distribuer ce programme à vos amis ou le mettre en téléchargement sur votre site, si vous le désirez.

Une chose importante à noter, cependant : veillez à copier, en même temps que votre programme, tout ce qui se trouve dans le dossier `dist`. Sans quoi, votre exécutable pourrait ne pas se lancer convenablement.

Le script **cxfreeze** est très pratique et suffit bien pour de petits programmes. Il com-

porte certaines options utiles que vous pouvez retrouver dans la documentation de **cx_Freeze**, accessible avec le code web suivant :

▷ Documentation cx_Freeze
Code web : 713172

Nous allons à présent voir une seconde méthode pour utiliser **cx_Freeze**.

Le fichier `setup.py`

La seconde méthode n'est pas bien plus difficile mais elle peut se révéler plus puissante à l'usage. Cette fois, nous allons créer un fichier `setup.py` qui se charge de créer l'exécutable de notre programme.

Un fichier `setup.py` basique contient ce code :

```
"""Fichier d'installation de notre script salut.py."""

from cx\_Freeze import setup, Executable

# On appelle la fonction setup
setup(
    name = "salut",
    version = "0.1",
    description = "Ce programme vous dit bonjour",
    executables = [Executable("salut.py")],
)
```

Tout tient dans l'appel à la fonction `setup`. Elle possède plusieurs arguments nommés :

- `name` : le nom de notre futur programme.
- `version` : sa version.
- `description` : sa description.
- `executables` : une liste contenant des objets de type `Executable`, type que vous importez de `cx_Freeze`. Pour se construire, celui-ci prend en paramètre le chemin du fichier `.py` (ici, c'est notre fichier `salut.py`).

Maintenant, pour créer votre exécutable, vous lancez `setup.py` en lui passant en paramètre la commande `build`.

Sur Windows, dans la ligne de commande : `C:\python32\python.exe setup.py build`.

Et sur Linux : `$ python3.2 setup.py build`.

Une fois l'opération terminée, vous aurez dans votre dossier un sous-répertoire `build`. Ce répertoire contient d'autres sous-répertoires portant différents noms en fonction de votre système.

Sur Windows, je trouve par exemple un dossier appelé `exe.win32-3.2`.

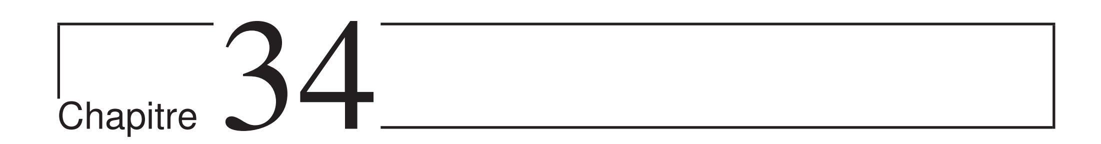
Dans ce dossier se trouve l'exécutable de votre programme et les fichiers dont il a besoin.

Pour conclure

Ceci n'est qu'un survol de **cx_Freeze**. Vous trouverez plus d'informations dans la documentation indiquée plus haut, si vous voulez connaître les différentes façons d'utiliser **cx_Freeze**.

En résumé

- **cx_Freeze** est un outil permettant de créer des programmes Python *standalone*.
- Un programme *standalone* signifie qu'il contient lui-même les dépendances dont il peut avoir besoin, ce qui rend sa distribution plus simple.
- **cx_Freeze** installe un script qui permet de créer nos programmes *standalone* très rapidement.
- On peut arriver à un résultat analogue en créant un fichier appelé traditionnellement `setup.py`.



Chapitre 34

De bonnes pratiques

Difficulté : 

Nous allons à présent nous intéresser à quelques **bonnes pratiques** de codage en Python.

Les conventions que nous allons voir sont, naturellement, des propositions. Vous pouvez coder en Python sans les suivre.

Toutefois, prenez le temps de considérer les quelques affirmations ci-dessous. Si vous vous sentez concernés, ne serait-ce que par une d'entre elles, je vous invite à lire ce chapitre :

- Un code dont on est l'auteur peut être difficile à relire si on l'abandonne quelque temps.
- Lire le code d'un autre développeur est toujours plus délicat.
- Si votre code doit être utilisé par d'autres, il doit être facile à reprendre (à lire et à comprendre).



Pourquoi suivre les conventions des PEP ?

Vous avez absolument le droit de répondre en disant que personne ne lira votre code de toute façon et que vous n'aurez aucun mal à comprendre votre propre code. Seulement, si votre code prend des proportions importantes, si l'application que vous développez devient de plus en plus utilisée ou si vous vous lancez dans un gros projet, il est préférable pour vous d'adopter quelques conventions clairement définies dès le début. Et, étant donné qu'il n'est jamais certain qu'un projet, même démarré comme un amusement passager, ne devienne pas un jour énorme, ayez les bons réflexes dès le début !

En outre, vous ne pouvez jamais être sûrs à cent pour cent qu'aucun développeur ne vous rejoindra, à terme, sur le projet. Si votre application est utilisée par d'autres, là encore, ce jour arrivera peut-être lorsque vous n'aurez pas assez de temps pour poursuivre seul son développement.

Quoi qu'il en soit, je vais vous présenter plusieurs conventions qui nous sont proposées au travers de PEP [1]. Encore une fois, il s'agit de propositions et vous pouvez choisir d'autres conventions si celles-ci ne vous plaisent pas.

La PEP 20 : tout une philosophie

La PEP 20, intitulée *The Zen of Python*, nous donne des conseils très généraux sur le développement. Vous pouvez la consulter grâce au code web suivant :

▷ PEP 20
Code web : 484505

Bien entendu, ce sont davantage des conseils axés sur « comment programmer en Python » mais la plupart d'entre eux peuvent s'appliquer à la programmation en général.

Je vous propose une traduction de cette PEP :

- *Beautiful is better than ugly* : le beau est préférable au laid ;
- *Explicit is better than implicit* : l'explicite est préférable à l'implicite ;
- *Simple is better than complex* : le simple est préférable au complexe ;
- *Complex is better than complicated* : le complexe est préférable au compliqué ;
- *Flat is better than nested* : le plat est préférable à l'imbriqué [2] ;
- *Sparse is better than dense* : l'aéré est préférable au compact ;
- *Readability counts* : la lisibilité compte ;
- *Special cases aren't special enough to break the rules* : les cas particuliers ne sont pas suffisamment particuliers pour casser la règle ;
- *Although practicality beats purity* : même si l'aspect pratique doit prendre le pas sur la pureté [3] ;

1. *Python Enhancement Proposal* : proposition d'amélioration de Python.
2. Moins littéralement, du code trop imbriqué (par exemple une boucle imbriquée dans une boucle imbriquée dans une boucle...) est plus difficile à lire.
3. Moins littéralement, il est difficile de faire un code à la fois fonctionnel et « pur ».

- *Errors should never pass silently* : les erreurs ne devraient jamais passer silencieusement ;
- *Unless explicitly silenced* : à moins qu'elles n'aient été explicitement réduites au silence ;
- *In the face of ambiguity, refuse the temptation to guess* : en cas d'ambiguïté, résistez à la tentation de deviner ;
- *There should be one – and preferably only one – obvious way to do it* : il devrait exister une (et de préférence une seule) manière évidente de procéder ;
- *Although that way may not be obvious at first unless you're Dutch* : même si cette manière n'est pas forcément évidente au premier abord, à moins que vous ne soyez Néerlandais ;
- *Now is better than never* : maintenant est préférable à jamais ;
- *Although never is often better than *right* now* : mais jamais est parfois préférable à *immédiatement* ;
- *If the implementation is hard to explain, it's a bad idea* : si la mise en œuvre est difficile à expliquer, c'est une mauvaise idée ;
- *If the implementation is easy to explain, it may be a good idea* : si la mise en œuvre est facile à expliquer, ce peut être une bonne idée ;
- *Namespaces are one honking great idea – let's do more of those* : les espaces de noms sont une très bonne idée (faisons-en plus !).

Comme vous le voyez, c'est une liste d'aphorismes très simples. Ils donnent des idées sur le développement Python mais, en les lisant pour la première fois, vous n'y voyez sans doute que peu de conseils pratiques.

Cependant, cette liste est vraiment importante et peut se révéler très utile. Certaines des idées qui s'y trouvent couvrent des pans entiers de la philosophie de Python.

Si vous travaillez sur un projet en équipe, un autre développeur pourra contester la mise en œuvre d'un extrait de code quelconque en se basant sur l'un des aphorismes cités plus haut.

Quand bien même vous travailleriez seul, il est toujours préférable de comprendre et d'appliquer la philosophie d'un langage quand on l'utilise pour du développement.

Je vous conseille donc de garder sous les yeux, autant que possible, cette synthèse de la philosophie de Python et de vous y référer à la moindre occasion. Commencez par lire chaque proposition. Les lignes sont courtes, prenez le temps de bien comprendre ce qu'elles veulent dire.

Sans trop détailler ce qui se trouve au-dessus (cela prendrait trop de temps), je signale à votre attention que plusieurs de ces aphorismes parlent surtout de l'allure du code. L'idée qui semble se dissimuler derrière, c'est qu'un code fonctionnel n'est pas suffisant : il faut, autant que possible, faire du « beau code ». Qui fonctionne, naturellement... mais ce n'est pas suffisant !

Maintenant, nous allons nous intéresser à deux autres PEP qui vous donnent des conseils très pratiques sur votre développement :

- la première nous donne des conseils très précis sur la présentation du code ;
- la seconde nous donne des conseils sur la documentation au cœur de notre code.

La PEP 8 : des conventions précises

Maintenant que nous avons vu des directives très générales, nous allons nous intéresser à une autre proposition d'amélioration, la PEP 8. Elle nous donne des conseils très précis sur la forme du code. Là encore, c'est à vous de voir : vous pouvez appliquer la totalité des conseils donnés ici ou une partie seulement. Voici le code web vous permettant d'accéder à cette PEP :

▷ PEP 8
Code web : 521237

Je ne vais pas reprendre tout ce qui figure dans cette PEP mais je vais expliquer la plupart des conseils en les simplifiant. Par conséquent, si l'une des propositions présentées dans cette section manque d'explications à vos yeux, je vous conseille d'aller faire un tour sur la PEP originale. Ce qui suit n'est pas une traduction complète, j'insiste sur ce point.

Introduction

L'une des convictions de Guido[4] est que le code est lu beaucoup plus souvent qu'il n'est écrit. Les conseils donnés ici sont censés améliorer la lisibilité du code. Comme le dit la PEP 20, *la lisibilité compte* !

Un guide comme celui-ci parle de cohérence. La cohérence au cœur d'un projet est importante. La cohérence au sein d'une fonction ou d'un module est encore plus importante.

Mais il est encore plus essentiel de savoir « quand » être incohérent (parfois, les conseils de style donnés ici ne s'appliquent pas). En cas de doute, remettez-vous en à votre bon sens. Regardez plusieurs exemples et choisissez celui qui semble le meilleur.

Il y a deux bonnes raisons de ne pas respecter une règle donnée :

1. Quand appliquer la règle rend le code moins lisible.
2. Dans un soucis de cohérence avec du code existant qui ne respecte pas cette règle non plus. Ce cas peut se produire si vous utilisez un module ou une bibliothèque qui ne respecte pas les mêmes conventions que celles définies ici.

Forme du code

- Indentation : utilisez 4 espaces par niveau d'indentation.
- Tabulations ou espaces : ne mélangez *jamais*, dans le même projet, des indentations à base d'espaces et d'autres à base de tabulations. À choisir, on préfère généralement les espaces mais les tabulations peuvent être également utilisées pour marquer l'indentation.

4. Guido Van Rossum, créateur et BDFL (*Benevolent Dictator For Life* : « dictateur bienveillant à vie ») de Python.

– Longueur maximum d'une ligne : limitez vos lignes à un maximum de 79 caractères. De nombreux éditeurs favorisent des lignes de 79 caractères maximum. Pour les blocs de texte relativement longs (docstrings, par exemple), limitez-vous de préférence à 72 caractères par ligne.
  Quand cela est possible, découpez vos lignes en utilisant des parenthèses, crochets ou accolades plutôt que l'anti-slash \.
  Exemple :

```
appel_d_une_fonction(parametre_1, parametre_2,
        parametre_3, parametre_4):
    ...
```

  Si vous devez découper une ligne trop longue, faites la césure *après* l'opérateur, pas avant.

```
# Oui
    un_long_calcul = variable + \
            taux * 100

# Non
    un_long_calcul = variable \
            + taux * 100
```

– Sauts de ligne : séparez par deux sauts de ligne la définition d'une fonction et la définition d'une classe.
  Les définitions de méthodes au cœur d'une classe sont séparées par une ligne vide.
  Des sauts de ligne peuvent également être utilisés, parcimonieusement, pour délimiter des portions de code
– Encodage : à partir de Python 3.0, il est conseillé d'utiliser, dans du code comportant des accents, l'encodage Utf-8.

Directives d'importation

– Les directives d'importation doivent préférentiellement se trouver sur plusieurs lignes. Par exemple :

```
import os
import sys
```

  plutôt que :

```
import os, sys
```

  Cette syntaxe est cependant acceptée quand on importe certaines données d'un module :

```
from subprocess import Popen, PIPE
```

– Les directives d'importation doivent toujours se trouver en tête du fichier, sous la documentation éventuelle du module mais avant la définition de variables globales ou de constantes du module.
– Les directives d'importation doivent être divisées en trois groupes, dans l'ordre :

1. les directives d'importation faisant référence à la bibliothèque standard ;
2. les directives d'importation faisant référence à des bibliothèques tierces ;
3. les directives d'importation faisant référence à des modules de votre projet.

Il devrait y avoir un saut de ligne entre chaque groupe de directives d'importation.

– Dans vos directives d'importation, utilisez des chemins absolus plutôt que relatifs. Autrement dit :

```
from paquet.souspaquet import module

# Est préférable à
from . import module
```

Le signe espace dans les expressions et instructions

– Évitez le signe espace dans les situations suivantes :
  – Au cœur des parenthèses, crochets et accolades :

```
# Oui
    spam(ham[1], {eggs: 2})

# Non
    spam( ham[ 1 ], { eggs: 2 } )
```

  – Juste avant une virgule, un point-virgule ou un signe deux points :

```
# Oui
    if x == 4: print x, y; x, y = y, x

# Non
    if x == 4 : print x , y ; x , y = y , x
```

  – Juste avant la parenthèse ouvrante qui introduit la liste des paramètres d'une fonction :

```
# Oui
    spam(1)

# Non
    spam (1)
```

  – Juste avant le crochet ouvrant indiquant une indexation ou sélection :

```
# Oui
    dict['key'] = list[index]

# Non
    dict ['key'] = list [index]
```

  – Plus d'un espace autour de l'opérateur d'affectation = (ou autre) pour l'aligner avec une autre instruction :

```
# Oui
    x = 1
    y = 2
    long_variable = 3

# Non
    x             = 1
    y             = 2
    long_variable = 3
```

– Toujours entourer les opérateurs suivants d'un espace (un avant le symbole, un après) :
  – affectation : `=`, `+=`, `-=`, etc. ;
  – comparaison : `<`, `>`, `<=`, ..., `in`, `not in`, `is`, `is not` ;
  – booléens : `and`, `or`, `not` ;
  – arithmétiques : `+`, `-`, `*`, etc.

```
# Oui
    i = i + 1
    submitted += 1
    x = x * 2 - 1
    hypot2 = x * x + y * y
    c = (a + b) * (a - b)

# Non
    i=i+1
    submitted +=1
    x = x*2 - 1
    hypot2 = x*x + y*y
    c = (a+b) * (a-b)
```



Attention : n'utilisez pas d'espaces autour du signe = si c'est dans le contexte d'un paramètre ayant une valeur par défaut (définition d'une fonction) ou d'un appel de paramètre (appel de fonction).

```
# Oui
    def fonction(parametre=5):
        ...
    fonction(parametre=32)

# Non
    def fonction(parametre = 5):
        ...
    fonction(parametre = 32)
```

– Il est déconseillé de mettre plusieurs instructions sur une même ligne :

```
# Oui
    if foo == 'blah':
        do_blah_thing()
    do_one()
```

```
    do_two()
    do_three()

# Plutôt que
    if foo == 'blah': do_blah_thing()
    do_one(); do_two(); do_three()
```

Commentaires



Les commentaires qui contredisent le code sont pires qu'une absence de commentaire. Lorsque le code doit changer, faites passer parmi vos priorités absolues la mise à jour des commentaires !

- Les commentaires doivent être des phrases complètes, commençant par une majuscule. Le point terminant la phrase peut être absent si le commentaire est court.
- Si vous écrivez en anglais, les règles de langue définies par Strunk and White dans « The Elements of Style » s'appliquent.
- À l'attention des codeurs non-anglophones : s'il vous plaît, écrivez vos commentaires en anglais, sauf si vous êtes sûrs à 120% que votre code ne sera jamais lu par quelqu'un qui ne comprend pas votre langue (ou que vous ne parlez vraiment pas un mot d'anglais !).

Conventions de nommage

Noms à éviter

N'utilisez jamais les caractères suivants de manière isolée comme noms de variables : `l` (L minuscule), `O` (o majuscule) et `I` (i majuscule). L'affichage de ces caractères dans certaines polices fait qu'ils peuvent être aisément confondus avec les chiffres `0` ou `1`.

Noms des modules et packages

Les modules et packages doivent avoir des noms courts, constitués de lettres minuscules. Les noms de modules peuvent contenir des signes `_` (souligné). Bien que les noms de packages puissent également en contenir, la PEP 8 nous le déconseille.

Noms de classes

Sans presque aucune exception, les noms de classes utilisent la convention suivante : la variable est écrite en minuscules, exceptée la première lettre de chaque mot qui la constitue. Par exemple : `MaClasse`.

Noms d'exceptions

Les exceptions étant des classes, elles suivent la même convention. En anglais, si l'exception est une erreur, on fait suivre le nom du suffixe `Error` (vous retrouvez cette convention dans `SyntaxError`, `IndexError`...).

Noms de variables, fonctions et méthodes

La même convention est utilisée pour les noms de variables (instances d'objets), de fonctions ou de méthodes : le nom est entièrement écrit en minuscules et les mots sont séparés par des signes soulignés (_). Exemple : `nom_de_fonction`.

Constantes

Les constantes doivent être écrites entièrement en majuscules, les mots étant séparés par un signe souligné (_). Exemple : `NOM_DE_MA_CONSTANTE`.

Conventions de programmation

Comparaisons

Les comparaisons avec des singletons (comme `None`) doivent toujours se faire avec les opérateurs `is` et `is not`, jamais avec les opérateurs `==` ou `!=`.

```
# Oui
    if objet is None:
        ...

# Non
    if objet == None:
        ...
```

Quand cela est possible, utilisez l'instruction `if objet:` si vous voulez dire `if objet is not None:`.

La vérification du type d'un objet doit se faire avec la fonction `isinstance` :

```
# Oui
    if isinstance(variable, str):
        ...

# Non
    if type(variable) == str:
        ...
```

Quand vous comparez des séquences, utilisez le fait qu'une séquence vide est `False`.

```
if liste: # La liste n'est pas vide
```

Enfin, ne comparez pas des booléens à `True` ou `False` :

```
# Oui
    if booleen: # Si booleen est vrai
        ...
    if not booleen: # Si booleen n'est pas vrai
        ...

# Non
    if booleen == True:
        ...

# Encore pire
    if booleen is True:
        ...
```

Conclusion

Voilà pour la PEP 8 ! Elle contient beaucoup de conventions et toutes ne figurent pas dans cette section. Celles que j'ai présentées ici, dans tous les cas, sont moins détaillées. Je vous invite donc à faire un tour du côté du texte original si vous désirez en savoir plus.

La PEP 257 : de belles documentations

Nous allons nous intéresser à présent à la PEP 257 qui définit d'autres conventions concernant la documentation via les docstrings. Consultez-la grâce au code web suivant :

▷ PEP 257
Code web : 959195

```
1 def fonction(parametre1, parametre2):
2     """Documentation de la fonction."""
```

La ligne 2 de ce code, que vous avez sans doute reconnue, est une docstring. Nous allons voir quelques conventions autour de l'écriture de ces docstrings (comment les rédiger, qu'y faire figurer, etc.).

Une fois de plus, je vais prendre quelques libertés avec le texte original de la PEP. Je ne vous proposerai pas une traduction complète de la PEP mais je reviendrai sur les points que je considère importants.

Qu'est-ce qu'une docstring ?

La docstring (chaîne de documentation, en français) est une chaîne de caractères placée juste après la définition d'un module, d'une classe, fonction ou méthode. Cette chaîne

de caractères devient l'attribut spécial `__doc__` de l'objet.

```
>>> fonction.__doc__
'Documentation de la fonction.'
>>>
```

Tous les modules doivent être documentés grâce aux docstrings. Les fonctions et classes exportées par un module doivent également être documentées ainsi. Cela vaut aussi pour les méthodes publiques d'une classe (y compris le constructeur `__init__`). Un package peut être documenté via une docstring placée dans le fichier `__init__.py`.

Pour des raisons de cohérence, utilisez toujours des guillemets triples `"""` autour de vos docstrings. Utilisez `"""chaîne de documentation"""` si votre chaîne comporte des anti-slash \.

On peut trouver les docstrings sous deux formes :

- sur une seule ligne ;
- sur plusieurs lignes.

Les docstrings sur une seule ligne

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Notes

- Les guillemets triples sont utilisés même si la chaîne tient sur une seule ligne. Il est plus simple de l'étendre par la suite dans ces conditions.
- Les trois guillemets `"""` fermant la chaîne sont sur la même ligne que les trois guillemets qui l'ouvrent. Ceci est préférable pour une docstring d'une seule ligne.
- Il n'y a aucun saut de ligne avant ou après la docstring.
- La chaîne de documentation est une phrase, elle se termine par un point `.`.
- La docstring sur une seule ligne *ne doit pas* décrire la signature des paramètres à passer à la fonction/méthode, ou son type de retour. N'écrivez pas :

  ```
  def fonction(a, b):
      """fonction(a, b) -> list"""
  ```

  Cette syntaxe est uniquement valable pour les fonctions C (comme les *built-ins*). Pour les fonctions Python, l'introspection peut être utilisée pour déterminer les paramètres attendus. L'introspection ne peut cependant pas être utilisée pour déterminer le type de retour de la fonction/méthode. Si vous voulez le préciser, incluez-le dans la docstring sous une forme explicite :

  ```
  """Fonction faisant cela et renvoyant une liste."""
  ```

  Bien entendu, « faisant cela » doit être remplacé par une description utile de ce que fait la fonction !

Les docstrings sur plusieurs lignes

Les docstrings sur plusieurs lignes sont constituées d'une première ligne résumant brièvement l'objet (fonction, méthode, classe, module), suivie d'un saut de ligne, suivi d'une description plus longue. Respectez autant que faire se peut cette convention : une ligne de description brève, un saut de ligne puis une description plus longue.

La première ligne de la docstring peut se trouver juste après les guillemets ouvrant la chaîne ou juste en-dessous. Dans tous les cas, le reste de la docstring doit être indenté au même niveau que la première ligne :

```
class MaClasse:
    def __init__(self, ...):
        """Constructeur de la classe MaClasse

        Une description plus longue...
        sur plusieurs lignes...

        """
```

Insérez un saut de ligne avant et après chaque docstring documentant une classe.

La docstring d'un module doit généralement dresser la liste des classes, exceptions et fonctions, ainsi que des autres objets exportés par ce module (une ligne de description par objet). Cette ligne de description donne généralement moins d'informations sur l'objet que sa propre documentation. La documentation d'un package (la docstring se trouvant dans le fichier `__init__.py`) doit également dresser la liste des modules et sous-packages qu'il exporte.

La documentation d'une fonction ou méthode doit décrire son comportement et documenter ses arguments, sa valeur de retour, ses effets de bord, les exceptions qu'elle peut lever et les restrictions concernant son appel (quand ou dans quelles conditions appeler cette fonction). Les paramètres optionnels doivent également être documentés.

```
def complexe(reel=0.0, image=0.0):
    """Forme un nombre complexe.

    Paramètres nommés :
    reel -- la partie réelle (0.0 par défaut)
    image -- la partie imaginaire (0.0 par défaut)

    """
    if image == 0.0 and reel == 0.0: return complexe_zero
    ...
```

La documentation d'une classe doit, de même, décrire son comportement, documenter ses méthodes publiques et ses attributs.

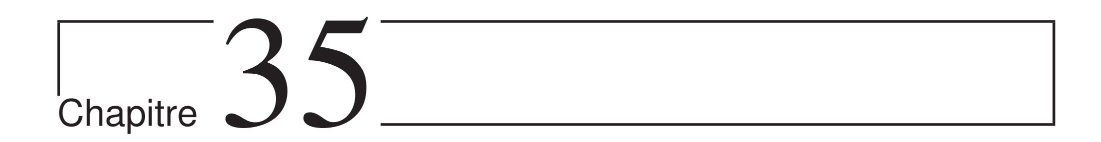
Le BDFL nous conseille de sauter une ligne avant de fermer nos docstrings quand elles sont sur plusieurs lignes. Les trois guillemets fermant la docstring sont ainsi sur une ligne vide par ailleurs.

```
def fonction():
```

```
    """Documentation brève sur une ligne.

    Documentation plus longue...

    """
```



Chapitre 35

Pour finir et bien continuer

Difficulté : 

La fin de ce cours sur Python approche. Mais si ce langage vous a plu, vous aimeriez probablement concrétiser vos futurs projets avec lui. Je vous donne ici quelques indications qui devraient vous y aider.

Ce sera cependant en grande partie à vous d'explorer les pistes que je vous propose. Vous avez à présent un bagage suffisant pour vous lancer à corps perdu dans un projet d'une certaine importance, tant que vous vous en sentez la motivation.

Nous allons commencer par voir quelques-unes des ressources disponibles sur Python, pour compléter vos connaissances sur ce langage.

Nous verrons ensuite plusieurs bibliothèques tierces spécialisées dans certains domaines, qui permettent par exemple de réaliser des interfaces graphiques.



Quelques références

Dans cette section, je vais surtout parler des ressources officielles que l'on peut trouver sur le site de Python. Pour vous y rendre, utilisez le code web suivant :

▷ Site officiel de Python
Code web : 338274

Il en existe bien entendu d'autres, certaines d'entre elles sont en français. Mais les ressources les plus à jour concernant Python se trouvent sur le site de Python lui-même.

En outre, les ressources mises à disposition sont clairement expliquées et détaillées avec assez d'exemples pour comprendre leur utilité. Elles n'ont qu'un inconvénient : elles sont en anglais. Mais c'est le cas de la majeure partie des documentations en programmation et il faudra bien envisager, un jour où l'autre, de s'y mettre pour aller plus loin !

La documentation officielle

Nous avons déjà parlé de la documentation officielle dans ces pages. Nous allons maintenant voir comment elle se décompose exactement.

Commencez par vous rendre sur le site de Python. Dans le menu de navigation, vous pourrez trouver plusieurs liens (notamment le lien de téléchargement, `DOWNLOAD`, sur lequel vous avez probablement cliqué pour obtenir Python). Il s'y trouve également le lien `DOCUMENTATION` et c'est sur celui-ci que je vous invite à cliquer à présent.

Dans la nouvelle page qui s'affiche figurent deux éléments intéressants :

- Sous le lien `DOCUMENTATION` du menu, il y a à présent un sous-menu contenant les liens `Current Docs`, `License`, `Help`, etc.
- La partie centrale de la page contient maintenant des informations sur les documentations de Python, classées suivant les versions. Par défaut, seules les deux versions les plus récentes de Python (dans les branches 2.X et 3.X) sont visibles mais vous pouvez afficher toutes les versions en cliquant sur le lien `the complete list of documentation by Python version`.

Nous allons d'abord nous intéresser au sous-menu.

Le wiki Python

Python propose un wiki en ligne qui centralise de nombreuses informations autour de Python. Si vous cliquez sur le lien `Wiki` dans le sous-menu, vous êtes conduits à une nouvelle page. Il n'existe pas de version traduite du wiki, vous devez donc demander à accéder à la page d'accueil en anglais.

Une fois là, vous avez accès à une vaste quantité d'informations classées par catégories. Je vous laisse explorer si vous êtes intéressés.

L'index des PEP (Python Enhancement Proposal)

Dans ce sous-menu, vous pouvez également trouver un lien intitulé `PEP Index`. Si vous cliquez dessus, vous accédez à un tableau, ou plutôt à un ensemble de tableaux reprenant les PEPs classées par catégories. Comme vous pouvez le constater, il y en a un paquet et, dans ce livre, je n'ai pu vous en présenter que quelques-unes. Libre à vous de parcourir cet index et de vous pencher sur certaines des PEP en fonction des sujets qui vous intéressent plus particulièrement.

La documentation par version

À présent, revenez sur la page de documentation de Python. Cliquez sur le lien correspondant à la version de Python installée sur votre machine (`Browse Python 3.2.1 Documentation` pour moi).

Sur la nouvelle page qui s'affiche sous vos yeux, vous trouvez les grandes catégories de la documentation. En voici quelques-unes :

- **Tutorial** : le tutoriel. Selon toute probabilité, les bases de Python vous sont acquises ; il est néanmoins toujours utile d'aller faire un tour sur cette page pour consulter la table des matières.
- **Library Reference** : la référence de la bibliothèque standard, nous reviendrons un peu plus loin sur cette page. Le conseil donné me paraît bon à suivre : *Keep this under your pillow*, c'est-à-dire, « gardez-la sous votre oreiller ».
- **Language Reference** : cette page décrit d'une façon très explicite la syntaxe du langage.
- **Python HOWTOs** : une page regroupant des documents d'aide traitant de sujets très précis, par exemple comment bien utiliser les sockets.

Vous pouvez aussi trouver un classement par index que je vous laisse découvrir. Vous pourrez y voir, notamment, le lien permettant d'afficher la table des matières complète de la documentation. Vous y trouverez également un glossaire, utile dans certains cas.

La référence de la bibliothèque standard

Vous vous êtes peut-être déjà rendus sur cette page. Je l'espère, en vérité. Elle comporte la documentation des types prédéfinis par Python, des fonctions *built-in* et exceptions, mais aussi des modules que l'on peut trouver dans la bibliothèque standard de Python. Ces modules sont classés par catégories et il est assez facile (et parfois très utile) de survoler la table des matières pour savoir ce que Python nous permet de faire sans installer de bibliothèque tierce.

C'est déjà pas mal, comme vous pouvez le voir !

Cela dit, il existe certains cas où des bibliothèques tierces sont nécessaires. Nous allons voir quelques-uns de ces cas dans la suite de ce chapitre, ainsi que quelques bibliothèques utiles dans ces circonstances.

Des bibliothèques tierces

La bibliothèque standard de Python comporte déjà beaucoup de modules et de fonctionnalités. Mais il arrive, pour certains projets, qu'elle ne suffise pas.

Si vous avez besoin de créer une application avec une interface graphique, la bibliothèque standard vous propose un module appelé **tkinter**. Il existe toutefois d'autres moyens de créer des interfaces graphiques, en faisant appel à des bibliothèques tierces.

Ces bibliothèques se présentent comme des packages ou modules que vous installez pour les rendre accessibles depuis votre interpréteur Python.



À l'heure où j'écris ces lignes, toutes les bibliothèques dont je parle ne sont pas nécessairement compatibles avec Python 3.X.

Les développeurs desdites bibliothèques ont généralement comme projet, à plus ou moins long terme, de passer leur code en Python 3.X. Si certains ont déjà franchi le pas, d'autres attendent encore et ce travail est plus ou moins long en fonction des dépendances de la bibliothèque.

Bref, tout cela évolue et si je vous dis que telle bibliothèque n'est pas compatible avec Python 3.X, il faudra entendre *pour l'instant*. À l'heure où vous lisez ces lignes, il est bien possible qu'une version compatible soit parue. Le changement se fait, lentement mais sûrement.

Ceci étant posé, examinons quelques bibliothèques tierces. Il en existe un nombre incalculable et, naturellement, je n'en présente ici qu'une petite partie.

Pour créer une interface graphique

Nous avons parlé de **tkinter**. Il s'agit d'un module disponible par défaut dans la bibliothèque standard de Python. Il se base sur la bibliothèque **Tk** et permet de développer des interfaces graphiques.

Il est cependant possible que ce module ne corresponde pas à vos besoins. Il existe plusieurs bibliothèques tierces qui permettent de développer des interfaces graphiques, parfois en proposant quelques bonus. En voici trois parmi d'autres :

PyQT : une bibliothèque permettant le développement d'interfaces graphiques, actuellement en version 4. En outre, elle propose plusieurs packages gérant le réseau, le SQL (bases de données), un kit de développement web... et bien d'autres choses. Soyez vigilants cependant : PyQt est distribuée sous plusieurs licences, commerciales ou non. Vous devrez tenir compte de ce fait si vous commencez à l'utiliser. La biliothèque est accessible via le code web suivant :

▷ PyQt
Code web : 230480

PyGTK : comme son nom l'indique, c'est une bibliothèque faisant le lien entre Py-

thon et la bibliothèque **GTK / GTK+**. Elle est distribuée sous licence LGPL et est accessible via le code web suivant :

▷ PyGTK
Code web : 116989

wx Python : une bibliothèque faisant le lien entre Python et la bibliothèque WxWidget, accessible via le code web suivant :

▷ wxPython
Code web : 223008

Ces informations ne vous permettent pas de faire un choix immédiat entre telle ou telle bibliothèque, j'en ai conscience. Aussi, je vous invite à aller jeter un coup d'œil du côté des sites de ces différents projets.

Ces trois bibliothèques ont l'avantage d'être multiplateformes et, généralement, assez simples à apprendre. En fonction de vos besoins, vous vous tournerez plutôt vers l'une ou l'autre, mais je ne peux certainement pas vous aider dans ce choix. Je vous invite donc à rechercher par vous-mêmes si vous êtes intéressés.

Dans le monde du Web

Il existe là encore de nombreuses bibliothèques, bien que je n'en présente ici que deux. Elles permettent de créer des sites web et concurrencent des langages comme PHP.

Django

La première, dont vous avez sans doute entendu parler auparavant, est Django. Vous pouvez y accéder grâce à ce code web :

▷ Django
Code web : 304222

Django est une bibliothèque, ou plutôt un *framework*, permettant de développer votre site dynamique en Python. Il propose de nombreuses fonctionnalités que vous trouverez, je pense, aussi puissantes que flexibles si vous prenez le temps de vous pencher sur le site du projet.

À l'heure où j'écris ces lignes, certains développeurs tentent de proposer des patches pour adapter Django à Python 3. L'équipe du projet, cependant, ne prévoit pas dans l'immédiat de développer de branche pour Python 3.

Si vous tenez réellement à Django et qu'aucune version stable n'existe encore sous Python 3 à l'heure où vous lisez ces lignes, je vous encourage à tester cette bibliothèque sous Python 2.X. Rassurez-vous, vous n'aurez pas à apprendre toute la syntaxe pour programmer dans ce langage : la liste des changements est très clairement affichée sur le site de Python et ceux-ci ne représentent pas un obstacle insurmontable pour qui est motivé !

CherryPy

CherryPy est une bibliothèque permettant de construire tout votre site en Python. Elle n'a pas la même vocation que Django puisqu'elle permet de créer la hiérarchie de votre site dynamiquement mais vous laisse libres des outils à employer pour faire quelque chose de plus complexe. Vous pouvez la trouver avec le code web suivant :

▷ CherryPy
Code web : 398287

Un peu de réseau

Pour finir, nous allons parler d'une bibliothèque assez connue, appelée **Twisted**.

▷ Site officiel de Twisted
Code web : 315304

Cette bibliothèque est orientée vers le réseau. Elle prend en charge de nombreux protocoles de communication réseau (TCP et UDP, bien entendu, mais aussi HTTP, SSH et de nombreux autres). Si vous voulez créer une application utilisant l'un de ces protocoles, Twisted pourrait être une bonne solution.

Là encore, je vous invite à jeter un coup d'œil sur le site du projet pour plus d'informations. À l'heure où j'écris, Twisted n'est utilisable que sous la branche 2.X de Python. Cependant, on peut trouver une future version, en cours d'élaboration, portant Twisted sur la branche 3.X.

Pour conclure

Ce ne sont là que quelques bibliothèques tierces, il en existe de nombreuses autres, certaines dédiées à des projets très précis. Je vous invite à faire des recherches plus avancées si vous avez des besoins plus spécifiques. Vous pouvez commencer avec la liste de bibliothèques qui se trouve ci-dessus, avec les réserves suivantes :

– Je ne donne que peu d'informations sur chaque bibliothèque et elles ne s'accordent peut-être plus avec celles disponibles sur le site du projet. En outre, la documentation de chaque bibliothèque reste et restera, dans tous les cas, une source plus sûre et actuelle.
– Ces projets évoluent rapidement. Il est fort possible que les informations que je fournis sur ces bibliothèques ne soient plus vraies à l'heure où vous lisez ces lignes. Pour mettre à jour ces informations, il n'y a qu'une seule solution imparable : allez sur le site du projet !

Une dernière petite parenthèse avant de vous quitter : je me suis efforcé de présenter, tout au long de ce livre, des données utiles et à jour sur le langage de programmation Python, dans sa branche 3.X. Il vous reste encore de nombreuses choses à découvrir sur le langage et ses bibliothèques, mais vous êtes désormais capables de voler de vos propres ailes. Bonne route ! ;-)

Index

Notes

Notes

Notes

Notes

Notes

Notes

Dépôt légal : septembre 2011
ISBN : 979-10-90085-03-9
Code éditeur : 979-10-90085
Imprimé en France

Achevé d'imprimer le 20 septembre 2011
sur les presses de Corlet Imprimeur (Condé-sur-Noireau)
Numéro imprimeur : 140422



Mentions légales :
Crédit photo Vincent Le Goff 4e de couverture : Fan Jiyong
Conception couverture : Fan Jiyong
Illustrations chapitres : Fan Jiyong

APPRENEZ À PROGRAMMER EN PYTHON

Vous n'y connaissez rien en programmation et souhaitez apprendre **un langage clair et intuitif** ?
Python est fait pour vous !
Vous découvrirez dans ce livre **conçu pour les débutants** tout ce dont vous aurez besoin pour programmer, des bases à la bibliothèque standard en passant par la programmation orientée objet !



Plus de **30 chapitres** de difficulté progressive
Des **exercices réguliers** sous forme de TP

Un cours conçu pour les débutants

- Aucun pré-requis, à part savoir allumer son ordinateur
- Une difficulté progressive pour ne perdre aucun lecteur en route
- Des exemples clairs et une pédagogie adaptée aux débutants

La programmation Python pas à pas

- Qu'est-ce que la programmation ? Quel langage choisir ? Pourquoi Python ?
- Installez Python et partez à la découverte du langage
- Découvrez les concepts de la programmation orientée objet
- Pratiquez grâce aux TP : développez une roulette de casino, un jeu du pendu…
- Initiez-vous aux interfaces graphiques avec Tkinter
- Apprenez à communiquer en réseau dans vos programmes Python
- Découvrez les bonnes pratiques afin d'améliorer vos codes

À qui ce livre est-il destiné ?

- Aux passionnés d'informatique qui veulent aller plus loin avec leur ordinateur
- Aux étudiants dans le domaine des nouvelles technologies qui recherchent un support de cours
- À toutes les personnes qui ont besoin de se former ou de se convertir à la programmation

À propos de l'auteur



Vincent Le Goff

Passionné d'informatique, il découvre au lycée la programmation en Python, un langage qu'il affectionne tout particulièrement pour son aspect simple et puissant.

Actuellement étudiant à IN'TECH INFO, une école supérieure d'ingénierie informatique, il se spécialise en « Systèmes et réseaux ». Sur son temps libre, il utilise son expérience du langage Python pour former les débutants à la programmation sur le Site du Zéro et participe également à plusieurs projets open source. Une belle réussite quand on sait que Vincent est non-voyant !

Ce livre est issu du Site du Zéro

Retrouvez dans ce livre les cours du Site du Zéro dans une édition revue et corrigée avec de nouveaux chapitres inédits du même auteur !

Téléchargez les codes sources en ligne grâce aux « codes web » inclus dans ce livre.

ISBN : 979-10-90085-03-9



Prix public : 25 € TTC

le Livre du Zéro

www.siteduzero.com

